Characterizing and Mitigating Virtual Machine Interference in Public Clouds

by

Yunjing Xu

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2014

Doctoral Committee:

Associate Professor Michael Donald Bailey, Co-Chair
Professor Farnam Jahanian, Co-Chair
Professor Brian D. Noble
Richard D. Schlichting, AT&T Labs Research
Professor Douglas E. Van Houweling

To my wife Zhuojun

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisors Michael Bailey and Farnam Jahanian. They walked me through this long journey as tutors and as friends. Whenever I became blocked, Michael's door was always open for me. I am also proud of being Michael's very first doctoral student. Moreover, Farnam made sure that I did not miss the big picture.

I would like to thank the rest of my doctoral committee, including Brian Nobel, Richard Schlichting, and Douglas Van Houweling. All of them have provided excellent feedback that shaped my dissertation in every aspect—from the big picture to the use of individual words. Brian was practically my third advisor. A large portion of my dissertation is based on the papers we wrote together.

I also want to thank the many great people that I have worked with throughout my graduate school years. Richard is not only my committee member, but also my internship supervisor, along with Kaustubh Joshi and Matti Hiltunen at AT&T Labs Research. The collaboration with them laid the foundation of my Ph.D. research. In addition, I want to thank Niels Provos and Lucas Ballard at Google, from whom I learned the invaluable experience of conducting research at the Internet scale.

Thanks to Timur Alperovich, Zhiyun Qian, Jie Yu, Lujun Fang, Eric Wustrow, Mona Attariyan, Mike Chow, Jon Oberheide, Evan Cooke, and my many friends at Michigan. The conversations with them were probably not the most productive things I could have done, but definitely the best part of my graduate school experience.

Most importantly, I want to thank my wife and my parents. This dissertation is dedicated to my beloved wife Zhuojun. I would not have had the determination to finish this journal without her constant support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

Table

LIST OF FIGURES

Figure

ABSTRACT

Characterizing and Mitigating Virtual Machine Interference in Public Clouds

by

Yunjing Xu

Co-Chairs: Michael Donald Bailey and Farnam Jahanian

This dissertation studies the mitigation of the performance and security interference between guest virtual machines (VMs) in public clouds. The goals are to characterize the impact of VM interference, uncover the root cause of the negative impact, and design novel techniques to mitigate such impact. The central premise of this dissertation is that by identifying the shared resources that cause the VM interference and by exploiting the properties of the workloads that share these resources with adapted scheduling policies, public cloud services can reduce conflicts of resource usage between guests and hence mitigate their interference. Current techniques for conflict reduction and interference mitigation overlook the virtualization semantic gap between the cloud host infrastructure and guest virtual machines and the unique challenges posed by the multi-tenancy service model necessary to support public cloud services.

This dissertation deals with both performance and security interference problems. It characterizes the impact of VM interference on inter-VM network latency using live measurements in a real public cloud and studies the root cause of the negative impact with controlled experiments on a local testbed. Two methods of improving the inter-VM network latency are explored. The first approach is a *guest-centric* solution that exploits the properties of application workloads to avoid interference without any support from the un-

derlying host infrastructure. The second approach is a *host-centric* solution that adapts the scheduling policies for the contented resources that cause the interference without guest cooperation. Similarly, the characteristics of cache-based cross-VM attacks are studied in detail using both live cloud measurements and testbed experiments. To mitigate this security interference, a *partition-based* VM scheduling system is designed to reduce the effectiveness of these cache-based attacks.

Thesis Statement: By adapting resource scheduling policies to the distinctive properties of latency-bound and throughput-bound workloads, it is possible to mitigate the performance and security interference between virtual machines that share hardware resources in public clouds.

CHAPTER 1

Introduction

With the emergence of cloud computing in the mid 2000s, computing resources became public utility—a concept that dates back to the early 60s [28]. Among cloud computing paradigms, Infrastructure-as-a-Service (IaaS) employs a pay-as-you-go model that allows anyone with a valid credit card to rent a large amount of computing resources from cloud data centers and only pay for what they use, without an upfront investment in hardware infrastructure.

Public IaaS clouds, such as Amazon Elastic Cloud Compute (EC2) [54], are often used to build Internet-scale Web applications, such as Netflix, Yelp, and Pinterest [73]. The impact of public clouds on the consumer Internet is therefore enormous. For example, as of April 2012, sites built on Amazon's cloud alone attract one third of all Internet users every day and contribute to more than 1% of all Internet consumer traffic [48].

A distinguishing feature of public clouds is *multi-tenancy*—hardware infrastructure is shared by various users of different organizations. Thus, instead of allowing direct hardware access, cloud providers use *virtualization* to give users access to computing resources in the form of virtual machines (VMs), while still keeping the full control of the underlying hardware infrastructure. Ideally, virtualization should give users the illusion of dedicated hardware access and provide strong performance and security isolation between the virtual machines that share physical machines, the data center network, or other layers of the cloud infrastructure, so that they cannot interfere with one another.

Unfortunately, such isolation is routinely violated because of the contention for the

shared resources that are multiplexed between guest VMs in public clouds [43]. VMs running different tasks may exhibit different resource access patterns. If VMs with conflicting resource access patterns are co-located on the same physical machines, they may cause *performance* interference with each other [47, 55]. For example, the performance of a workload with temporal locality in its memory access pattern relies heavily on the efficiency of various levels of CPU caches, but its neighboring VMs may be running workloads that cause frequent cache eviction to force repeated main memory access for the same contents [2]. Worse, such performance interference can even be abused by malicious neighboring VMs to cause *security* problems. Potential threats include performance degradation attacks [76] and cross-VM information leakage [66, 97].

Mitigating the performance and security interference between guest virtual machines in public clouds is challenging because virtualization carries a *semantic gap* [17] between the guests who manage application workloads and the hosts who manage the cloud infrastructure. Optimizations at one layer are made without understanding the mechanisms or even intentions at another, and they tend to operate at cross purposes. For instance, from the perspective of cloud guests, the extent of resource contention is determined by the resource schedulers that are host-controlled and operating below all guest VMs, while from the host's perspective, applications' resource usage patterns may affect their scheduling policies, but only guest VMs have the knowledge of such patterns.

The goals of this dissertation are to characterize the impact of VM interference on performance and security and then design novel techniques to mitigate the negative impact. The key idea is to identify the shared resources that caused the VM interference and adapt the resource scheduling policies by exploiting the characteristics of guest application workloads. To mitigate the performance interference, two approaches are designed to function from different sides of the virtualization abstraction. A *guest-centric* solution exploits the difference in the qualitative properties of guest workloads to avoid excess resource conflicts without any support from the underlying host infrastructure, whereas *a host-centric* solution adapts the scheduling policies for the contented resources that cause the interference, without guest cooperation. Additionally, a partition-based VM scheduling system is designed from the perspective of the host infrastructure to mitigate the security interference.

To realize this key idea, the first step is to characterize the impact of performance and security interference and study their root causes. The potential impact of performance interference may exist for the throughput and latency of network I/O, disk I/O and computational jobs. As network I/O latency becomes increasingly important for Internet-scale user-facing applications [5, 95], this dissertation characterizes the impact and root cause of the performance interference on the inter-VM network latency. Meanwhile, researchers have designed various cross-VM attacks that are mounted by abusing shared processors, memory, or I/O subsystems [66, 60, 76, 85, 97]. This dissertation also characterizes the security interference by exploring the details of two cross-VM attacks—a performance degradation attack and a covert channel attack—that abuse the shared CPU cache. These attacks are of particular interest due to the prevalence of processor sharing in public clouds.

The characterization studies show that both the performance and security interference are consequences of the resource contention between guest VMs; this dissertation therefore explores novel techniques to reduce conflicts of resource usage and mitigate VM interference. By definition, avoiding hardware sharing completely would eliminate all possible contention [43], but it also defeats the economic model of cloud computing. Instead, to mitigate the performance interference, while still preserving the benefits of resource multiplexing, a restricted form of sharing is first designed to seek processor sharing with only compatible workloads to therefore reduce the conflicts. This approach only requires knowledge about guest application workloads above the virtualization semantic gap—it allows guest VMs to reduce network latency without any infrastructure changes.

In addition, this dissertation presents a technique that adapts the resource scheduling policies below the virtualization semantic gap to mitigate the impact of performance interference on inter-VM network latency without guest cooperation. This is a holistic design that not only alleviates the performance interference caused by the contention of shared processors, but it also reduces the conflicts of queueing space usage that exist in the host network stack and data center switches, respectively, using the same design principles. The key idea is to use shortest remaining time first scheduling [69]. This technique trades the throughput of large tasks for reduced latency of small tasks, but it is possible to do so without undue harm to throughput.

Moreover, to mitigate the security interference, a partition-based VM scheduling system is designed from the perspective of cloud service providers. By scheduling latency-bound and throughput-bound VMs separately on disjointed sets of processors and assigning them with scheduling policies tuned to their respective workload characteristics, this dissertation shows that the negative impact of two cache-based cross-VM attacks—a performance degradation attack and an information leakage attack—can be significantly reduced. Importantly, by building this system on the unique properties of public cloud services, the security benefits are obtained without introducing performance overhead to benign workloads. Instead, the performance of both latency-bound and throughput-bound workloads are improved by using the same partition-based scheduling model.

The overall goal of this dissertation is to understand the impact of virtual machine interference from both the performance and security perspectives, to analyze their root cause, and to design novel techniques to mitigate the negative impact. In summary, this dissertation makes the following contributions:

- **Characterization of inter-VM network latency in public clouds.** The network latency between hosts within a data center is critical to the performance of large-scale distributed systems, especially those with large fan-outs [5, 6, 95]. Using live measurements in Amazon EC2, which is a leading public cloud provider [51], and testbed experiments, we characterize the inter-VM network latency in virtualized cloud environments. The study shows that the impact of interference on network latency exists across the entire cloud host infrastructure—from the virtualization layer, through the host network stack, to the network switches. The mean latency is found to be more than twice as much as the baseline that has no interference, and the tail latency is more than an order of magnitude higher. Worse, compared to its counterpart in dedicated data centers [5], the tail latency can still be two to four times as bad. Demonstrated by controlled experiments, the root cause of the problem is the unchecked contention of shared resources, including processors and the queueing space in both physical hosts and data center switches, between guest virtual machines, and the primary reason for the long tail latency is the contention of shared processors between incompatible workloads—latency-bound ones versus CPU-bound ones.

- **Design of a guest-centric solution to avoid long tail latency.** As a first step to mitigate the impact of interference on inter-VM network latency, a system called Bobtail is designed to proactively detect the contention of shared processors between incompatible workloads and seek to share only between compatible ones, as the incompatible sharing is the primary reason for the large latency tail. This is a guest-centric design that allows cloud guests to exploit workload placement by leveraging the knowledge of their workload properties without extra support from the underlying host infrastructure. Using Bobtail, common communication patterns benefit from reductions of up to 40% in 99.9th percentile response times.

- **Design of a host-centric solution to mitigate all three latency problems.** To tackle all three latency problems revealed by the characterization study, a host-centric solution is designed to mitigate the performance interference by scheduling the contended resources using the Shortest Remaining Time First principle, without causing undue harm to throughput. Importantly, this goal can be achieved below the virtualization abstraction by cloud providers without requiring or trusting guest cooperation. Experimental and simulation results show that this host-centric solution can reduce median latency of small flows by 40%, with improvements in the tail of almost 90%, while reducing throughput of large flows by less than 3%.

- **Characterization of cache-based cross-VM attacks.** Cache sharing between guest VMs gives rise to security threats against neighboring VMs on the same physical machine. Therefore, we characterize the impact of security interference by studying two cache-based cross-VM attacks. Using controlled experiments on a testbed, the relationship between VMs' workload patterns and the effectiveness of the cache-based performance degradation attack against CPU-bound tasks is revealed. In addition, the threat of a cross-VM covert channel that abuses the contention of shared L2 cache is explored, and the limits of this threat are demonstrated by providing a quantification of the channel bit rates and an assessment of its ability to do harm. Through progressively refining models of this covert channel, from the derived maximums, to implementable channels on the testbed, and finally in Amazon EC2 itself, our study

shows how a variety of factors impact the ability to create effective channels, and how this channel is only practical to leak small secrets like private keys. Importantly, frequent VM preemption is demonstrated as the key to the effectiveness of both attacks.

- **Design of a host-centric solution to mitigate security interference.** To mitigate the impact of cache-based cross-VM attacks, a partition-based VM scheduling system is designed from the perspective of cloud providers. Because frequent VM preemption is critical to the effectiveness of the target attacks, this system separates latency-bound VMs from throughput-bound VMs, and it schedules them on disjointed sets of processors with scheduling policies tuned to their respective workload characteristics. This scheduling scheme restricts cache sharing to only compatible workloads. Therefore, it can mitigate the cache interference caused by the performance degradation attack and reduce the amount of information that can be leaked via shared cache. Importantly, the resulting system also improves the performance of benign workloads due to controlled resource sharing, instead of incurring undue overhead.

The rest of the dissertation is organized as follows: Chapter 2 reviews the background and related work relevant to the topic of this dissertation. The impact of VM interference on inter-VM network latency and its root causes are presented in Chapter 3. Chapters 4 and 5 discuss the solutions that mitigate the impact performance interference on inter-VM network latency from above and below the virtualization semantic gap, respectively. Chapter 6 characterizes the security impact of two cache-based cross-VM attacks. Chapter 7 discusses a partition-based VM scheduling system designed to mitigate the security interference on shared CPU cache. Finally, Chapter 8 concludes the dissertation and discusses future directions.

CHAPTER 2

Background and Related Work

In this chapter, we review background information and the related work regarding cloud hardware infrastructure and the resource management software. We choose Amazon's Elastic Compute Cloud as the subject of study since it is one of the largest public cloud offerings [48] and the market leader [51]. In addition, we use the Xen hypervisor [11], which is known to support EC2's infrastructure [78], to review virtualization technologies.

2.1 Public Cloud Performance

Public clouds employ an infrastructure-as-a-service model that allows developers to rent VM instances in a pay-as-you-go manner. Amazon's Elastic Compute Cloud (EC2) [54] is a major public cloud service provider used by many developers to build Internet-scale applications. EC2 consists of multiple geographically-separated *regions* around the world. In EC2's terminology, each region contains several *availability zones*, or AZs, that are physically isolated and have independent failure probabilities; therefore, an AZ can be considered a data center (DC). Throughout this dissertation, AZ and DC are used interchangeably unless noted otherwise. Hosts within the same AZ, or in different AZs within the same region, are connected by a high-speed private network. Meanwhile, hosts within different regions are connected by the public Internet. In other words, the traffic sent from EC2's east region to its west region is managed and billed in the same way as the traffic sent from a third-party cloud service. A VM in EC2 is called an *instance*. There are different

types of instances based upon size, performance characteristics, and cost. Depending on types, a physical machine may be shared by more than one instance. Such configuration information can be inferred using live measurements [66].

Various performance properties of EC2 have been explored. Wang *et al.* showed that the network performance of EC2 is much more variable than that of non-virtualized clusters due to virtualization and processor sharing [78]. Similarly, Schad *et al.* found a bimodal performance distribution with high variance for most of their metrics related to CPU, disk I/O, and network [68]. Barker *et al.* also quantified the jitter of CPU, disk, and network performance in EC2 and its impact on latency-sensitive applications [12]. A. Li *et al.* compared multiple cloud providers, including EC2, using many types of workloads and claimed that there is no single winner across all metrics [52]. Ou *et al.* considered hardware heterogeneity within EC2, and they noted that within a single instance type and availability zone, the variation in performance for CPU-intensive workloads can be as high as 60% [62]. Following the same observation, Farley *et al.* conducted a study of placement gaming that allows customers to exploit the performance variability in public clouds to lower the cost of running their own workloads [27]. In this dissertation, we study the problems of inter-VM network latency caused by VM interference and design novel techniques from the perspectives of both cloud providers and their customers to mitigate the problems.

2.2 Public Cloud Security

In public clouds, resource sharing can be abused and give rise to various security problems. This dissertation studies two specific types of attacks—performance degradation attacks [76, 67] and information leakage attacks [66]. A performance degradation attack is an extension of the performance interference caused by resource contention. Attackers can profile the activities of their neighboring VMs and intensify the contention of their shared resources, on purpose, to reduce the performance of their neighbors.

An information leakage attack allows one VM to learn information from its neighbors via shared resources in an unintended way. In 1973, Lampson discussed a classification of the ways in which information can be transferred between programs (i.e., legitimate,

storage, and covert channels) and defined covert channels as "those not intended for information transfer at all, such as the service program's effect on the system load." [49] Two classes of covert channels have emerged—those based on storage and those based on timing. In storage attacks, existing fields, memory locations, etc., are used to (secretly) encode information. For example, unused fields in network protocols can be used to convey information in an unintended way [1, 3, 29].

Timing-based information leakage attacks are more sophisticated because the information is encoded by varying the timing of events in a system. Thus, the receiver of a timing-based information channel must understand the original encoding scheme in order to obtain the actual information. For example, information channels constructed using the time intervals between network packets fall into this category [15]. Leveraging cache-based timing channels to extract cryptographic keys has been also studied extensively [46, 64]. Ristenpart *et al.* are the first to study the cross-VM information leakage problem in public clouds [66]. They profiled the timing in accessing various resources shared by VMs on the same physical machine and explored the timing difference to learn information secretively across the virtual machine boundary. The security measurement in this dissertation expands the scope of the pioneering work for this threat to show how a variety of factors impact the ability to create effective timing channels.

2.3 Virtualization and Resource Scheduling

Virtualization technology is the cornerstone of public cloud services. It allows tenants of various organizations to share the provider-controlled data center infrastructure. Because EC2 uses the Xen hypervisor [11] with various unknown customizations to support its service [78], the internals of Xen are discussed in this dissertation when a detailed understanding of the virtualization technology is required.

The Xen hypervisor is an open source virtualization solution for various platforms including x86, x86_64, IA64, and ARM [11]. It supports a virtualization technique called *paravirtualization*, in addition to full virtualization. Compared to full virtualization, paravirtualization does not require hardware support, so it can be used on legacy hardware,

while still providing acceptable performance. On the other hand, paravirtualization requires modifications to the guest operating system kernel to directly interact with the underlying hypervisor via *hypercalls*, which are analogous to syscalls in the operating system. The Xen hypervisor only provides resource protection and scheduling functionalities and the hypercall interface. Each guest operating system running on top of Xen is called a *domain*. Virtual machine management and device drivers are delegated to a special privileged domain called *dom0*. Other non-privileged domains, known as *domU*, can only access devices indirectly via dom0. Other virtualization platforms may have a host OS to process device I/O requests for guest VMs. Because Xen's dom0 and the host OS are functionally equivalent for the discussion in this dissertation, we use these two terms interchangeably despite their technical difference.

To manage hardware resources, Xen uses a credit-based VM scheduler [88]. By default, it allocates 30ms of CPU time (time slice) to each virtual CPU (VCPU). This allocation is decremented in 10ms intervals. Once a VCPU has exhausted its credit, it is not allowed to use CPU time unless there is no other VCPU with credit remaining, as any VCPU with credit remaining has a higher priority than any without. A runnable VCPU may preempt a running VCPU if the former has a higher priority after the latter has run for more than 1ms (rate limit). In addition, as described by Dunlap [25], a lightly-loaded VCPU with excess credit may enter the `BOOST` state, which allows a VM to automatically receive first execution priority when it wakes from an I/O interrupt. VMs in the same `BOOST` state run in FIFO order. Even with this optimization, Xen's credit scheduler is known to be unfair to latency-sensitive workloads [78, 25] and susceptible to performance interference [61, 47, 80].

To solve these problems, various characteristics of this credit scheduler have been examined, including scheduler configurations [61] and the source of overhead incurred by virtualization on the network layer [80]. Many designs have been proposed to improve Xen's default VM scheduler [31, 25, 44, 39, 18, 91, 90], and we compare these solutions in § 5.1. In addition to improving the VM scheduler itself, Wood *et al.* created a framework for the automatic migration of virtual machines between physical hosts in Xen when resources become a bottleneck [83]. Mei *et al.* also pointed out that a strategic co-placement of different workload types in a virtualized data center will improve performance for both cloud con-

sumers and cloud providers [55]. One of the interference mitigation techniques proposed in this dissertation also leverages a similar strategy that enables latency-aware workload placement. Importantly, such placement can be achieved by cloud customers themselves rather than requiring any support from the service providers.

CHAPTER 3

Characterization of Inter-VM Network Latency

In this chapter, we study the performance interference between guest virtual machines (VMs) in virtualized data centers. Among various performance metrics, the end-to-end delay becomes increasingly important to the user experience for Internet-scale applications [5, 95]. Thus, we measure the impact of the interference on inter-VM network latency in Amazon's EC2, which is the market leader of public Infrastructure-as-a-Service (IaaS) clouds [51], and analyze the root cause of the problem.

This chapter is comprised of three parts. The first part describes the results of live latency measurements in several different EC2 data centers. While the mean latency we observe is not out of the ordinary, its distribution has both a more significant jitter and a longer tail than that observed in dedicated data centers. Importantly, the long tail phenomenon is a property of nodes rather than topology or network traffic; it is pervasive throughout EC2 data centers, and it is reasonably persistent. The tail of inter-VM latency is of particular concern for applications relying on many nodes to process large data sets at less than human-scale response times. For example, constructing a single page view for such applications may require contacting hundreds of services [23], and a lag in response time from any one of them can result in significant end-to-end delays [5].

The second part of the chapter studies the root cause of the long tail latency problem in EC2 using controlled experiments on a local testbed. While Wang *et al.* report that network latency in EC2 is highly variable and they speculate that virtualization and processor sharing make up the root cause [78], our results suggest that processor sharing under vir-

tualization is *not sufficient* to cause the long tail problem by itself. Instead, it is the VM scheduler that fails to control the interference between the VMs that contend for the shared processor resources when they are running incompatible workloads.

Finally, the third part of this chapter shows that VM scheduling is not the only source of latency found in virtualized data centers and other latency problems exist that are actually caused by the *network*. Using controlled experiments, we demonstrate the impact of switch queueing delays and host network queueing delays, which are caused by the unchecked resource contention of the respective queueing spaces. Importantly, these three latency problems exist in different layers of the virtualized data center infrastructure, and their impact on inter-VM network latency are independent of each other.

3.1 EC2 Live Measurements

In this part of the study, we describe a five-week measurement study of network latency in several different EC2 data centers in its east region. The focus of the study is on the tail of round-trip latency due to its disproportionate impact on user experience. Other studies have measured network performance in EC2, but they often use metrics like mean and variance to show jitter in network and application performance [78, 68]. While these measurements are useful for high-throughput applications like MapReduce [22], worst-case performance matters much more to applications like the Web that require excellent user experience [95]. Because of this, researchers use the RTTs at the 99th and 99.9th percentiles to measure flow tail completion times in dedicated data centers [5, 95].

3.1.1 Measurement Methodology

Alizadeh *et al.* show that the internal infrastructure of Web applications is based primarily on TCP [5]. But instead of using raw TCP measurement, we use a TCP-based RPC framework called `Thrift`. `Thrift` is popular among Web companies like Facebook [71] and delivers a more realistic measure of network performance at the application level. To measure application-level round-trip-times (RTTs), we time the completion of synchronous

RPC calls—`Thrift` adds about 60μs of overhead when compared to TCP SYN/ACK based raw RTT measurement. In addition, we use established TCP connections for all measurement, so the overhead of the TCP three-way handshake is not included in the RTTs.

3.1.2 Tail Latency Characterization

We tested network latency in EC2's US east region for five weeks. Figure 3.1 shows CDFs for both a combination of small, medium, and large instances and for discrete sets of those instances. While (a) and (b) show aggregate measurements both within and across availability zones (AZs), (c) shows discrete measurements for three instance types within a specific AZ.

In Figure 3.1(a), we instantiated 20 instances of each type for each plot, either within a single AZ or across two AZs in the same region. We observe that median RTTs within a single AZ, at $\sim$0.6ms, compare well to those in dedicated data centers at $\sim$0.4ms [5, 95], even though our measurement method adds 0.06ms of overhead. Inter-AZ measurements show a median RTT of under 1ms. However, distances between pairs of AZs may vary; measurements taken from another pair of AZs show a median RTT of around 2ms.

Figure 3.1(b) shows the 99th to 100th percentile range of (a) across all observations. Unfortunately, its results paint a different picture of latency measurements in Amazon's data centers. The 99.9th percentile of RTT measurements is *twice as bad* as the same metric in dedicated data centers [5, 95]. Individual nodes can have 99.9th percentile RTTs up to four times higher than those seen in such centers. Note that this observation holds for both curves; no matter whether the measurements are taken in the same data center or in different ones, the 99.9th percentiles are almost the same.

Medium, large, and extra large instances ostensibly offer better performance than their small counterparts. As one might expect, our measurements show that extra large instances do not exhibit the extra long tail problem ($<$0.9ms for the 99.9th percentile); but surprisingly, as shown in Figure 3.1(c), medium and large instances are susceptible to the problem. In other words, the extra long tail is not caused by a specific type of instance: all instance types shown in (c) are equally susceptible to the extra long tail at the 99.9th percentile. Note

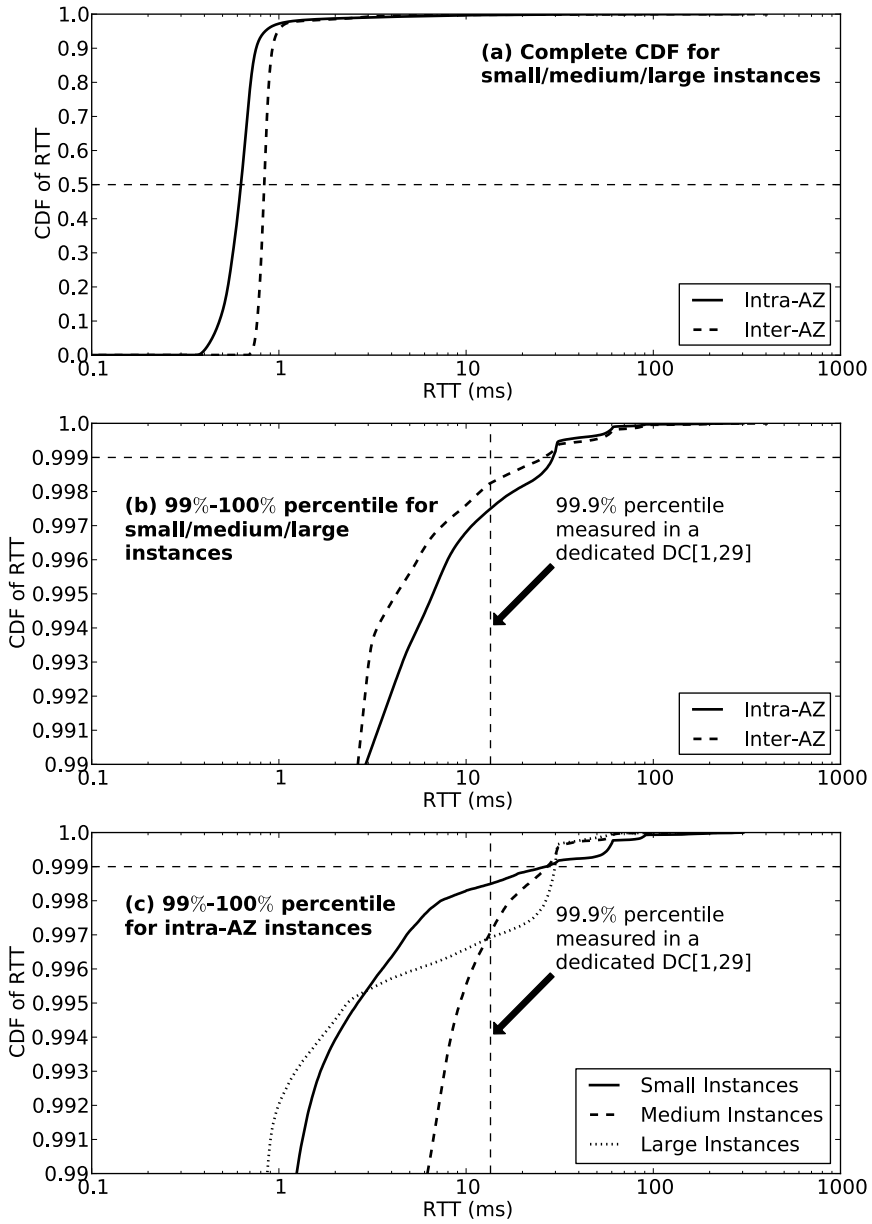


Figure 3.1: CDF of RTTs for various sized instances, within and across AZs in EC2, compared to measurements taken in dedicated data centers [5, 95]. While the median RTTs are comparable, the 99.9th percentiles in EC2 are twice as bad as in dedicated data centers. This relationship holds for all types of EC2 instances plotted.

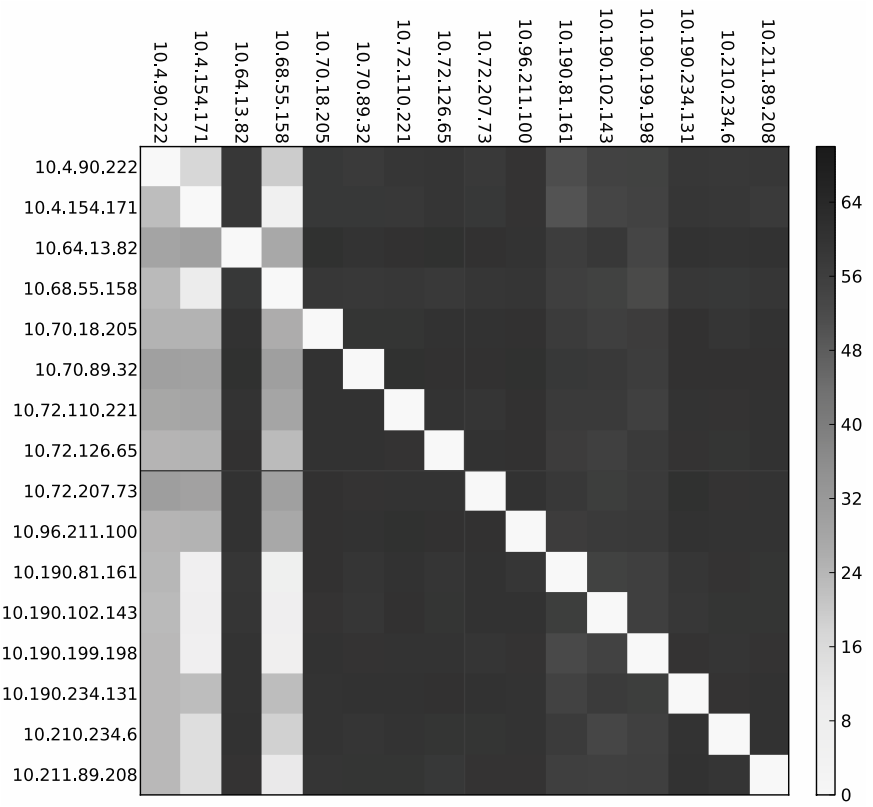


Figure 3.2: Heat map of the 99.9th percentile of RTTs, shown for 16 small pairwise instances in milliseconds. Bad instances, represented by dark vertical bands, are bad consistently. This suggests that the long tail problem is a property of specific nodes instead of the network.

that all three lines in the figure intersect at the 99.9th line with a value of around 30ms. The explanation of this phenomenon becomes evident in the discussion of the root cause of the long tail problem in § 3.2.

To explore other factors that might create extra long tails, we launch 16 instances within the same AZ and measure the *pairwise* RTTs between each instance. Figure 3.2 shows measurement results at the 99.9th percentile in milliseconds. Rows represent source IP addresses, while columns represent destination IP addresses.

Were host location on the network affecting long tail performance, we would see a symmetric pattern emerge on the heat map, since network RTT is a symmetric measurement. Surprisingly, the heat map is asymmetric—there are vertical bands which do not correspond to reciprocal pairings. To a large degree, the destination host controls whether a long tail exists. In other words, *the extra long tail problem in cloud environments is a property*

of nodes, rather than the network.

The data shown in Figure 3.2 is not entirely bleak: there are *both* dark and light bands, so tail performance between nodes varies drastically. Commonly, RPC servers are allowed only 10ms to return their results [5]. Therefore, we refer to nodes that fulfill this service as *good* nodes, which appear in Figure 3.2 as light bands; otherwise, they are referred to as *bad* nodes. Under this definition, we find that RTTs at the 99.9th percentile can vary by *up to an order of magnitude* between good nodes and bad nodes. In particular, the bad nodes we measured can be two times worse than those seen in dedicated DCs [5, 95] for the 99.9th percentile. This is because the latter case's latency tail is caused by network congestion, whose worst case impact is bounded by the egress queue size of the bottleneck switch port, but the latency tail problem we study here is a property of nodes, and its worst case impact can be much larger than that caused by network queueing delay. This observation will become more clear when we discuss the root cause of the problem in § 3.2.

To determine whether bad nodes are a pervasive problem in EC2, we spun up 300 small instances in each of four AZs in the US east region. We measured all the nodes' RTTs and found 40% to 70% bad nodes within three of the four AZs.

Interestingly, the remaining AZ sometimes does not return bad nodes; nevertheless, when it does, it returns 40% to 50% bad nodes. We notice that this AZ spans a smaller address space of only three /16 subnets compared to the others, which can span tens of /16 subnets. Also, its available CPU models are, on average, newer than those found in any of the other AZs; Ou *et al.* present similar findings [62], so we speculate that this data center is newly built and loaded more lightly than the others. We will discuss this issue further in conjunction with the root cause analysis in § 3.2.

We also want to explore whether the long latency tail we observe is a persistent problem, because it is a property defined by node conditions rather than transient network conditions. We conducted a five week experiment comprised of two sets of 32 small instances: one set was launched in equal parts from two AZs, and one set was launched from all four AZs. Within each set, we selected random pairs of instances and measured their RTTs throughout the five weeks. We observed how long instances' properties remain static—either good or bad without change—to show the persistence of our measurement results.

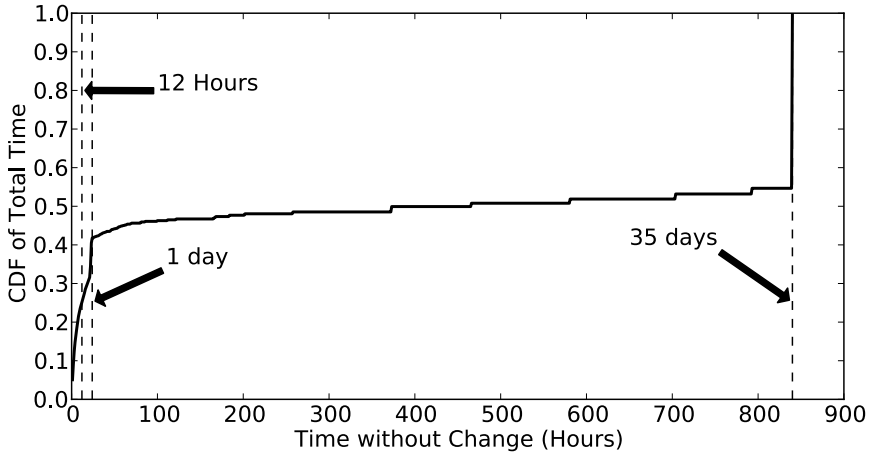


Figure 3.3: CDF for the time periods during which instances do not switch status between good and bad. This shows that properties of instances generally persist.

Figure 3.3 shows a CDF of these stable time periods; persistence follows if a large percentage of total instance time in the experiment is comprised of large time periods. We can observe that almost 50% of the total instance time no change has been witnessed, 60% of time involves at most one change per day, and 75% of time involves at most one change per 12 hours. This result shows that the properties of long tail network latency are generally persistent.

The above observation should be noted by the following: *every night*, *every instance* we observe in EC2 experiences an abnormally long latency tail for several minutes at midnight Pacific Time. For usually bad instances this does not matter; however, usually good instances are forced to change status at least once a day. Therefore, the figures we state above can be regarded as overestimating the frequency of changes. It also implies that the 50% instance time during which no change has been witnessed belongs to bad instances.

3.2 Root Cause Analysis

We know that the latency tail in EC2 is two to four times worse than that in dedicated data centers, and that as a property of nodes instead of the network it persists. In this part of the study, we answer the question of what is the root cause of the long tail latency problem in EC2 by conducting controlled experiments on a local testbed.

Generally speaking, the interference between neighboring VMs in the presence of processor sharing is likely to be the root cause of the problem [78] because the long tail latency is a property of nodes. However, the coexistence of good and bad instances suggests that processor sharing under virtualization is *not sufficient* to cause the long tail problem by itself. The results of our controlled experiments suggest that the impact of VM interference on network latency only manifests itself when the VMs sharing processors are running incompatible workloads; they are either latency-sensitive or CPU-intensive. Note that the information about the internals of the Xen hypervisor and its VM scheduler is required to understand the details of the controlled experiments and it is covered in § 2.3.

3.2.1 Varying Workload Mix

To demonstrate the relationship between workload incompatibility and the impact of VM interference, we conduct five experiments by vary the workload mix for the VMs co-located on the same physical machine. On a four-core workstation running Xen 4.1, dom0 is pinned to two cores while guest VMs use the rest. In all experiments, five identically configured domUs share the remaining two physical cores; they possess equal weights of up to 40% CPU utilization each. Therefore, though domUs may be scheduled on either physical core, none of them can use more than 40% of a single core even if there are spare cycles. To the best of our knowledge, this configuration is the closest possible to what EC2 *small* instances use. Note that starting from Xen 4.2, a configurable rate limit mechanism is introduced to the credit scheduler [88]. In its default setting, a running VM cannot be preempted if it has run for less than 1ms. To obtain a result comparable to the one in this section using Xen 4.2 or newer, the rate limit needs to be set to its minimum of 0.1ms.

For this set of experiments, we vary the workload types running on five VMs sharing the local workstation. In the first experiment, we run the `Thrift` RPC server to participate in a latency-sensitive workload in all five guest VMs; we use another non-virtualized workstation in the same local network to make RPC calls to all five servers, once every two milliseconds, for 15 minutes. During the experiment, the local network is never congested. In the next four experiments, we replace the RPC servers on the guest VMs with a CPU-

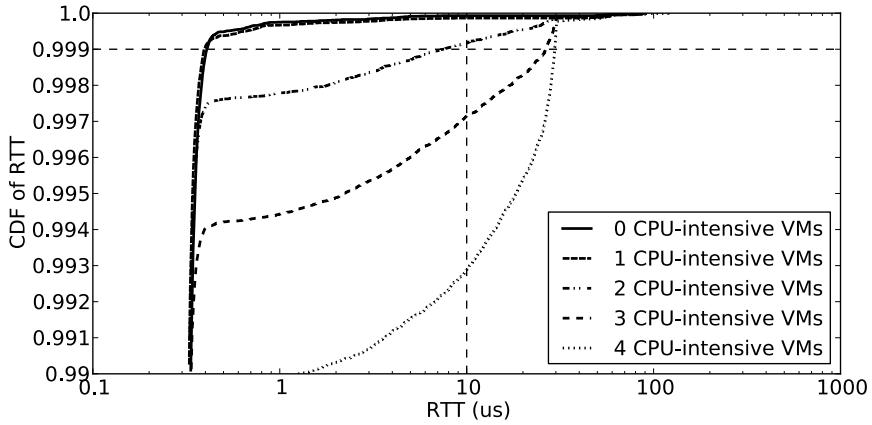


Figure 3.4: CDF of RTTs for a VM within controlled experiments, with an increasing number of co-located VMs running CPU-intensive workloads. Sharing does not cause extra long latency tails as long as physical cores outnumber CPU-intensive VMs, but once this condition no longer holds, the long tail emerges.

intensive workload, one at a time, until four guest VMs are CPU-intensive and the last one, called the *victim VM*, remains latency-sensitive.

Figure 3.4 shows the CDF of our five experiments' RTT distributions from the 99th to the 100th percentile for the victim VM. While four other VMs also run latency-sensitive jobs (zero VMs run CPU-intensive jobs), the latency tail up to the 99.9th percentile remains under 1ms. If one VM runs a CPU-intensive workload, this result does not change. Notably, even when the victim VM *does share* processors with one CPU-intensive VM and three latency-sensitive VMs, the extra long tail problem is *nonexistent*.

However, the 99.9th percentile becomes *five times* larger once two VMs run CPU-intensive jobs. This still qualifies as a good node under our definition ($<$10ms), but the introduction of even slight network congestion could change that. To make matters worse, RTT distributions increase further as more VMs become CPU-intensive. Eventually, the latency-sensitive victim VM behaves just like the bad nodes we observe in EC2.

The results of the controlled experiments assert that virtualization and processor sharing are not sufficient to cause high latency effects across the entire tail of the RTT distribution; therefore, much of the blame rests upon co-located workloads. We show that having one CPU-intensive VM is acceptable; why does adding one more suddenly make things five times worse?

There are two physical cores available to guest VMs; if we have one CPU-intensive VM, the latency-sensitive VMs can be scheduled as soon as they need to be, while the single CPU-intensive VM occupies the other core. Once we reach two CPU-intensive VMs, it becomes possible that they occupy both physical cores concurrently while the victim VM has an RPC request pending. Unfortunately, the `BOOST` mechanism does not appear to let the victim VM preempt the CPU-intensive VMs often enough. Resulting from these unfortunate scenarios is an extra long latency distribution. In other words, *sharing does not cause extra long latency tails as long as physical cores outnumber CPU-intensive VMs; once this condition no longer holds, the long tail emerges.*

3.2.2 Varying CPU Usage

The preceding controlled experiments demonstrate that Xen's processor scheduler fails to control the interference between the VMs that are running latency-sensitive and CPU-intensive workloads on shared processors, and such workload mix causes the long tail latency problem; but a question remains: will all CPU-intensive workloads have the same impact? In fact, we notice that if the co-located CPU-intensive VMs in the controlled experiments always use 100% CPU time, the latency-sensitive VM *does not* suffer from the long tail problem—its RTT distribution is similar to the one without co-located CPU-intensive VMs; the workload we use in the preceding experiments actually uses about 85% CPU time. This phenomenon can be explained by the design of the `BOOST` mechanism. Recall that a VM waking up due to an interrupt may enter the `BOOST` state if it has credits remaining. Thus, if a VM doing mostly CPU-bound operations decides to accumulate scheduling credits, e.g., by using the `sleep` function call, it will also get `BOOST`ed after the `sleep` timer expires. Then, it may monopolize the CPU until its credits are exhausted without being preempted by other `BOOST`ed VMs, some of which may be truly latency-sensitive.

In other words, the `BOOST` mechanism is only effective against the workloads that use almost 100% CPU time because such workloads exhaust their credits easily and `BOOST`ed VMs can then preempt them whenever they want. To study the impact of lower CPU usage, we conduct another controlled experiment by varying the CPU usage of the CPU-

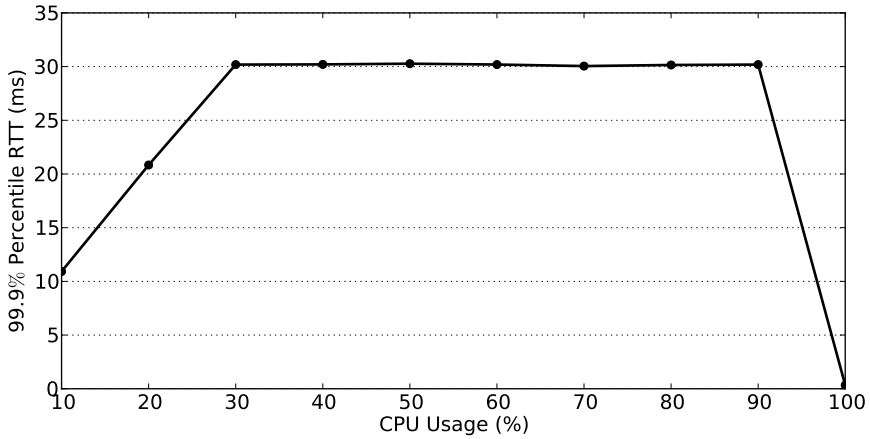


Figure 3.5: The relationship between the 99.9th percentile RTT for the latency-sensitive workload and the CPU usage of the CPU-bound workload in the neighboring VM.

intensive workload from 10% to 100% and measuring the 99.9th percentile RTT of the latency-sensitive workload. We control the CPU usage using a command-line utility called `cpulimit` on a process that otherwise uses 100% CPU time; `cpulimit` pauses the target process periodically to adjust its average CPU usage. In addition, based on what we learned from the first set of controlled experiments, we only need to use one latency-sensitive VM to share a single CPU core with one CPU-intensive VM and allocate 50% CPU time to each one respectively.

Figure 3.5 shows the relationship between the 99.9th percentile RTT of the latency-sensitive workload and the CPU usage of the CPU-bound workload in the neighboring VM. Surprisingly, the latency tail is over 10ms even with 10% CPU usage, and starting from 30%, the tail latency is almost constant until 100%. This is because by default `cpulimit` uses a 10ms granularity: given X% CPU usage, it makes the CPU-intensive workload work Xms and pause (100-X)ms in each 100ms window. Thus, when $X < 30$, the workload yields the CPU every Xms, so the 99.9th percentiles for the 10% and 20% cases are close to 10ms and 20ms, respectively; for $X \geq 30$, the workload keeps working for at least 30ms. Recall that the default time slice of the credit scheduler is 30ms, so the CPU-intensive workload cannot keep running for more than 30ms and we see the flat line in Figure 3.5. It also explains why the three curves in Figure 3.1(c) intersect at the 99.9th percentile line. The takeaway is that even if a workload uses as little as 10% CPU time on average, it still can cause a long latency tail to neighboring VMs by using large bursts of CPU cycles (e.g.,

10ms). In other words, average CPU usage does not capture the intensity of a CPU-bound workload; *it is the length of the bursts of CPU-bound operations that matters*.

Now that we understand the root cause, we will examine an issue stated earlier: one availability zone in the US east region of EC2 has a higher probability of returning good instances than the other AZs. If we break down VMs returned from this AZ by CPU model, we find a higher likelihood of newer CPUs. These newer CPUs should be more efficient at context switching, which naturally shortens the latency tail, but what likely matters more is newer CPUs' possessing six cores instead of four, as in older CPUs that are more common in the other three data centers. One potential explanation for this is that the EC2 instance scheduler may not consider CPU model differences when scheduling instances sensitive to delays. Then, a physical machine with four cores is much more likely to be saturated with CPU-intensive workloads than a six-core machine. Hence, a data center with older CPUs is more susceptible to the problem. Despite this, our root cause analysis always applies, because we have observed that both good and bad instances occur regardless of CPU model; differences between them only change the likelihood that a particular machine will suffer from the long tail problem.

3.3 More Latency Problems

The analysis in the last section demonstrates that the long tail latency problem is a property of nodes not the network. However, are there any other latency problems that are in fact caused by the network? In this section, we show that besides the VM scheduler, there exists at least two other latency sources in the virtualized data center infrastructure and both of them incur *network queueing delay* to the packets between guest VMs.

3.3.1 The Big Picture

Figure 3.6 shows three possible latency sources in the virtualized data center infrastructure. The VM scheduling delay—a property of nodes—is the subject of the measurement and analysis in the preceding sections. Meanwhile, the queueing delays found in the host

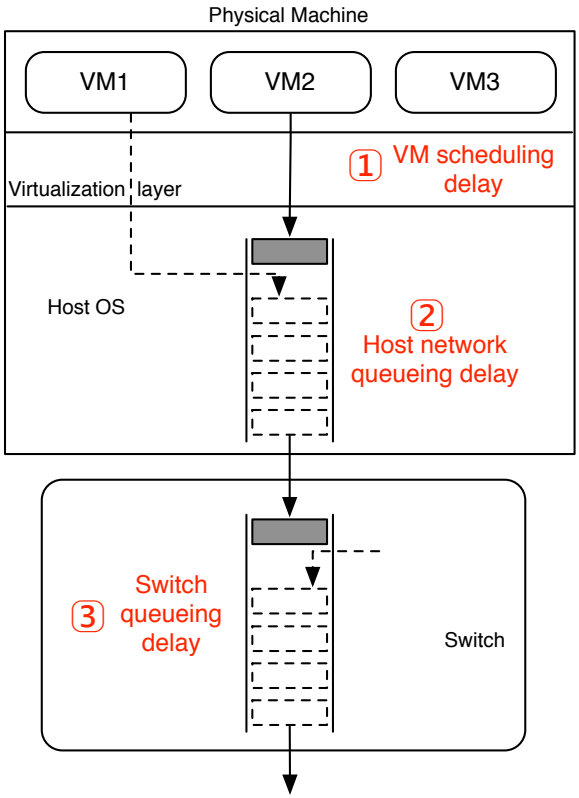


Figure 3.6: Three possible latency sources.

network stack and the data center switches are caused by the network itself. To understand them, consider a simple scenario where a client VM sends queries to a server VM for its responses. The time from the client to send a query and receive a complete response is defined as flow completion time (FCT).

(1) VM scheduling delay. When query packets arrive at the physical host running the server VM, that VM is notified of the reception. But the server VM cannot process the packets until scheduled by the hypervisor; this gap is called scheduling delay [39]. § 3.2 shows that while such delay is usually less than one millisecond, it could suffer from a long tail problem. A similar problem may also exist for applications running on bare-metal operating systems, but the added virtualization layer substantially exacerbates its impact [78].

(2) Host network queueing delay. After the server VM processes the query and sends a response, the response packets first go through the host network stack, which processes I/O requests on behalf of all guest VMs. The host network stack is another source of excessive

latency because it has to fulfill I/O requests for multiple VMs, which may contend to fill up the queues in the host network stack. In fact, reducing queueing delay in the kernel network stack has been a hot topic in the Linux community, and considerable advancements have been made [37, 57]. However, these existing mechanisms are designed without virtualization in mind; the interaction between the host network stack and guest VMs compromises the efficacy of these advancements.

(3) Switch queueing delay. Response packets on the wire may experience switch queueing delay on congested links in the same way as they do in dedicated data centers. Measurement studies show that the RTTs seen on a congested link in dedicated data centers vary by two orders of magnitude [5, 95]; virtualized data centers are no exception. What makes this problem worse in public clouds is that guest VMs on the same host may contend for limited bandwidth available to that host without the knowledge of each other. To continue our example, if the client VM is sharing hardware with a neighbor that receives large bursts of traffic, the response arriving to the client VM may experience queueing delay on its access link, even though the client VM itself is only using a small fraction of the bandwidth assigned to it.

In our example, the queueing delay for the host network stack and switches can occur for both query and response messages, while the VM scheduling delay almost always happens to query messages. In the next two subsections, we demonstrate the two queueing delay problems, respectively.

3.3.2 Latency in the Data Center Network

We demonstrate the queueing delay problem found in the data center network as we can measure it in EC2. The problem in the host network stack is illustrated in the next subsection; it requires to vary host kernel configurations on a testbed. In addition, we also compare the impact of switch queueing delay with that of VM scheduling delay because we can observe both problems on live EC2 instances.

Figure 3.7 shows the setup of our EC2 measurements. We measure the FCT between a client VM and two server VMs of the same EC2 account. In addition, a second EC2

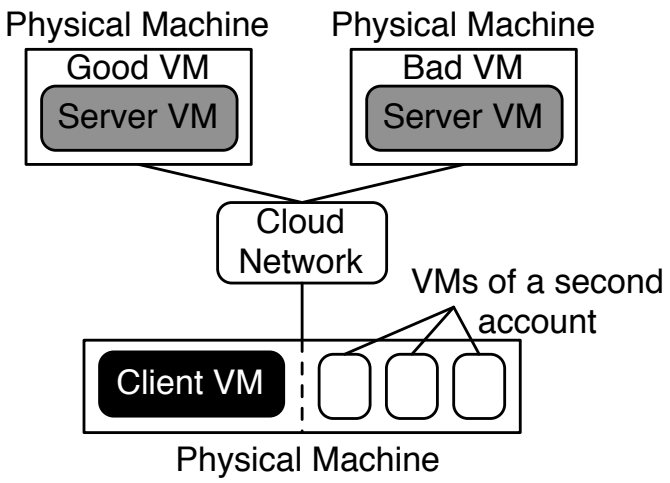


Figure 3.7: The setup of the EC2 measurements.

account is used to launch several VMs co-located with the client VM on the same host; these VMs are used to congest the access link shared with the client VM. All VMs used in this experiment are standard medium EC2 instances.

For switch queueing delay, we use the VMs of the second account to show that shared network links can be congested by an arbitrary EC2 customer without violating EC2's terms of use. Because it is hard to control the congestion level in the core EC2 network, our demonstration aims to congest the shared access link of the client VM. To do so, we launch 100 VMs with the second account and keep the ones that are verified to be co-located with the client VM on the same host. The techniques for verifying co-location are well studied [66, 92], so we omit the details.

With enough co-located VMs, we can congest the access link of the client VM by sending bulk traffic to the co-located VMs of the second account from other unrelated VMs. This means that the client VM may suffer from a large switch queueing delay caused by its neighbors on the same host, even though the client VM itself is only using a small fraction of its maximum bandwidth (1Gbps). During the measurement, we observe that the EC2 medium instances have both egress *and* ingress rate limit of 1Gbps, and the underlying physical machines in EC2 appear to have multiple network interfaces shared by their hosted VMs; therefore, we need at least three co-located VMs from the second account to obtain the congestion level needed for the demonstration.

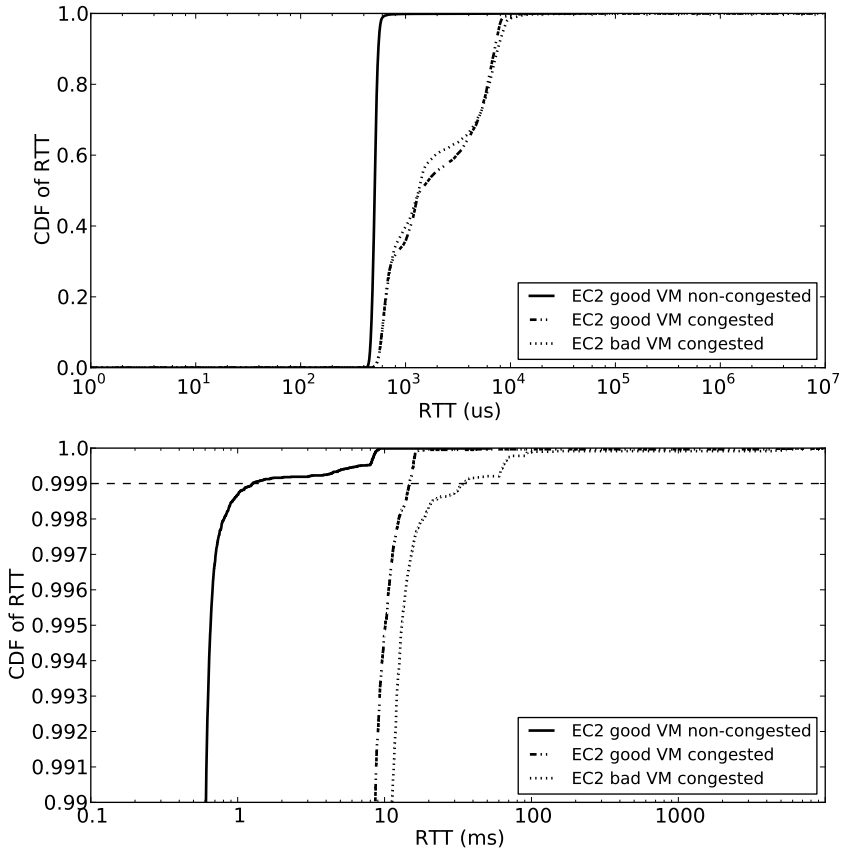


Figure 3.8: An EC2 experiment showing switch queueing delay (congested vs. non-congested) and VM scheduling delay (bad VM vs. good VM). Network congestion impacts both the body and tail distribution, while VM scheduling delay mostly impacts the tail at a larger scale.

In addition, to demonstrate VM scheduling delay, we launch several server VMs and measure their FCT distribution from the same client VM. The expectation is that the FCT for "Bad VMs" VMs have much larger tail latency than for "Good VMs", if the former are sharing CPU cores with certain VMs running CPU-bound jobs on the same host.

Figure 3.8 depicts the measurement results. There are three scenarios with progressively worsening FCT distributions to show the impact of individual latency sources: one with no delay, one with switch queueing delay only, and one with both switch queueing delay and VM scheduling delay. The top figure shows the entire FCT distribution, and the bottom one shows the distribution from the 99th to 100th percentile. Switch queueing delay has a large impact on both the body and the tail of the distribution, while VM scheduling delay affects mostly the tail of the distribution at a larger scale.

Scenarios	50th	90th	99th	99.9th
Good VM non-congested	1X	1X	1X	1X
Good VM congested	2.7X	12.9X	14.2	11.7X
Bad VM congested	2.6X	13.7X	18.5	27.7X

Table 3.1: The relative impact of switch queueing delay (congested vs. non-congested) and VM scheduling delay (bad VM vs. good VM).

To make things clearer, Table 3.1 summarizes the relative impact of both latency sources on different parts of the FCT distribution. The key observation is that while switch queueing delay alone can increase the latency by 2X at the 50th percentile and cause a 10X increase at the tail, VM scheduling delay can cause another 2X increase at the 99.9th percentile *independently*.

3.3.3 Latency in the Host Network Stack

In this subsection, we demonstrate the impact of the queueing delay found in the host network stack using a testbed. The host network stack has at least two queues for packet transmission: one in the kernel network stack that buffers packets from the TCP/IP layer and another in the network interface card (NIC) that takes packets from the first queue and sends them on the wire. NICs may contain many transmission queues, but we only consider the simple case for this demonstration.

A packet may experience queueing delay in the host network stack if either transmission queue is blocked by large bursts of packets from different VMs. This problem is not specific to virtualization-enabled hosts; a bare-metal Linux OS serving multiple applications can exhibit similar behaviors. Thus, we first discuss two features introduced in Linux 3.3 that tackle this problem, and then we explain why the interaction between the host network stack and guest VMs compromises the efficacy of these features.

Herbert *et al.* designed a new device driver interface called Byte Queue Limits (BQL) to manage NICs' transmission queues [37]. Without BQL, the length of NICs' transmission queues was measured by the number of packets, which are variable in size; so, the queueing delay for NICs is often unpredictable. BQL instead measures queue length by the number of bytes in the queue, which is a better predictor of queueing delay. To reduce queueing

delay while still maintaining high link utilization with BQL, one can now limit the queue length to the bandwidth-delay product [8] (e.g., for a 1Gbps link with 300us RTT, a queue of 37,500 bytes will suffice).

Now that NICs' transmission queues are properly managed, the queueing delay problem is pushed to the software queue in the kernel. Nichols *et al.* designed a packet scheduler called "CoDel" that schedules packets based on the amount of time a packet has spent in the queue and then compares it to a dynamically-calculated target of queueing delay [57]. If queued packets have already spent too much time in the queue, the upper layer of the network stack is notified to slow down, regardless of the queue occupancy. Linux's default behavior is to use a large drop tail queue (e.g., 1,000 packets), which incurs a large average queueing delay.

To use BQL and CoDel effectively, it is considered a best practice [86] to disable the packet segmentation offload (TSO and GSO) features offered by the NIC hardware. With TSO and GSO enabled, a packet in either of the transmission queues can be up to 64KB. Compared to Ethernet's MTU of 1,500 bytes, such large packets would again make queue length unpredictable. By disabling TSO and GSO, packets are segmented in software before joining the transmission queues allowing BQL and CoDel to have fine-grained control.

We set up controlled experiments to show the effectiveness of BQL and CoDel by congesting the host network queue and also to demonstrate why they are not sufficient in the presence of virtualization. The testbed has three physical machines running unmodified Xen 4.2.1 and Linux 3.6.6, and they are connected to a single switch. To observe host queueing delay, physical machine A runs two VMs, A1 and A2, that serve as senders. Another two VMs, B1 and C1, run in the remaining two physical machines, B and C, and serve as receivers. In this configuration, we make sure that none of the VMs would suffer from VM scheduling delay or switch queueing delay.

In the experiment, A2 `ping`s C1 10,000 times every 10ms to measure network latency. The traffic between A1 and B1 causes congestion. There are three scenarios:

1. Congestion Free: A1 and B1 are left idle.
2. Congestion Enabled: A1 sends bulk traffic to B1 without BQL or CoDel.
3. Congestion Managed: A1 sends bulk traffic to B1 with BQL and CoDel enabled.

Scenarios	50th	99th	99.9th
Congestion Free	0.217	0.235	0.250
Congestion Enabled	16.83	21.57	21.73
Congestion Managed	0.808	1.21	1.26

Table 3.2: Ping latency (ms). Despite the improvement using BQL and CoDel, congestion in the host network stack still increases the latency by four to six times.

For cases 2 and 3, A1 sends traffic to B1 using `iperf`, which saturates the access link of physical machine A.

Table 3.2 shows the distribution of the `ping` latency in milliseconds. Without BQL or CoDel, the contention for host network resources alone increases the latency by almost two orders of magnitude. While BQL and CoDel can significantly reduce the latency, the result is still four to six times as large when compared to the baseline. § 5.2.2 explains why the interaction between the host network stack and guest VMs is the source of such latency.

3.4 Summary

In this chapter, we study the performance interference between virtual machines by characterizing its impact on the inter-VM network latency. Using live measurements in EC2, we demonstrate that such interference exacerbates the long tail problem of network latency by a factor of two to four. Notably, we find that the long tail latency problem is a property of nodes, not the network, and it is pervasive throughout EC2 and persistent over time. Using controlled experiments, we show that Xen's processor scheduler does not always fail to control the interference on network latency; the failure happens when co-scheduling CPU-bound and latency-sensitive tasks that contend for the shared processors.

In addition, we also demonstrate that the VM scheduler is not the only place that manifests an excessive latency problem. The contention for the queueing spaces in data center switches and the host network stack can also incur large delays to inter-VM communication, and this is actually a problem of the *network*. Importantly, the impact of these latency sources are independent of each another: switch queueing delays can increase tail latency by over 10 times; together with VM scheduling delays, they become more than 20 times worse, while host network queueing delays also worsen the latency tail by four to six times.

CHAPTER 4

A Guest-Centric Approach to Avoid the Long Latency Tails

In this chapter, we design a guest-centric solution to mitigate the impact of performance interference on inter-VM network latency. Chapter 3 shows that virtualized data centers suffer from at least three latency problems whose impact is independent of each other. This chapter discusses a solution for one of these problems—the long tail latency caused by the co-scheduling of CPU-bound and latency-sensitive workloads on shared processors.

Because the measurement results indicate that EC2 contains both good and bad VM instances in terms of network tail latency, we design Bobtail [94] to find the good instances for running latency-sensitive applications. Specifically, the good instances are the ones that are sharing processors with only compatible workloads or those that do not share at all, while the bad instances are the ones that are co-scheduled with incompatible workloads. Therefore, cloud customers can then deploy their latency-sensitive applications only on the good VM instances to expect more predictable network performance.

Bobtail employs a guest-centric approach that only requires guests to know whether their workloads are latency-sensitive or CPU-bound. Cloud customers can use Bobtail as a library to decide on which instances to run their latency-sensitive workloads without any changes to the cloud infrastructure. While Bobtail does not improve the VM isolation directly, it mitigates the performance interference—common communication patterns benefit from reductions of up to 40% in 99.9th percentile response times.

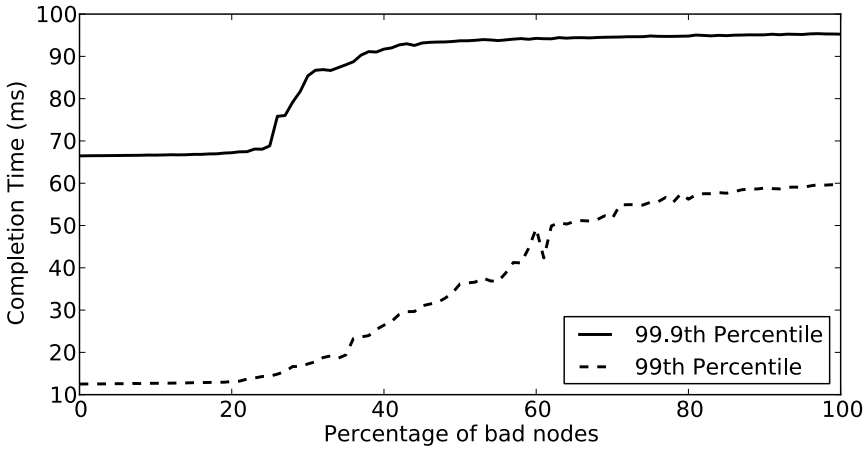


Figure 4.1: Impact of bad nodes on the flow tail completion times of the sequential model. Bobtail can expect to reduce tail flow completion time even when as many as 20% of nodes are bad.

4.1 Potential Benefits

To understand both how much improvement is possible and how hard it would be to obtain, we measured the impact of bad nodes for common communication patterns: sequential and partition-aggregation [95]. In the sequential model, an RPC client calls some number of servers in series to complete a single, timed observation. In the partition-aggregation model, an RPC client calls all workers in parallel for each timed observation.

For the sequential model, we simulate workflow completion time by sampling from the measured RTT distributions of good and bad nodes. Specifically, every time, we randomly choose one node out of N RPC servers to request 10 flows serially, and we repeat this $2,000,000$ times. Figure 4.1 shows the 99th and 99.9th percentile values of the workflow completion time, with an increasing number of bad nodes among a total of 100 instances.

Interestingly, there is no difference in the tails of overall completion times when as many as 20% of nodes are bad. But the difference in flow tail completion time between 20% bad nodes and 50% bad nodes is severe: flow completion time increases by *a factor of three* at the 99th percentile, and a similar pattern exists at the 99.9th percentile with a smaller difference. This means Bobtail is allowed to make mistakes—even if up to 20% of the instances picked by Bobtail are actually bad VMs, it still helps reduce flow completion time when compared to using random instances from EC2. Our measurements suggest that

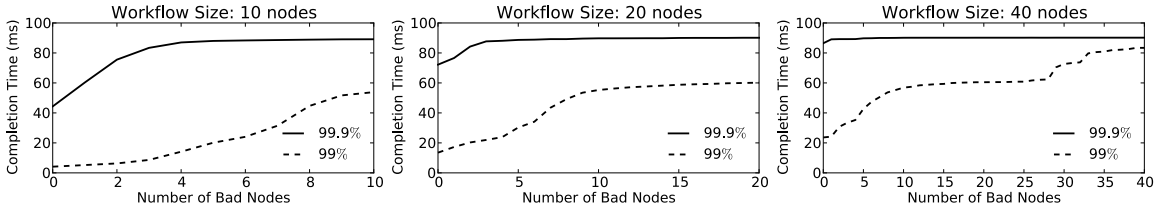


Figure 4.2: Impact of bad nodes on the tail completion time of the partition-aggregation model with 10, 20, and 40 nodes involved in the workloads. At modest scales, with fan-outs of 10 or even 20 nodes, there are substantial gains to be realized by avoiding bad nodes.

receiving 50% bad nodes from EC2 is not uncommon.

Figure 4.2 shows the completion time of the partition-aggregation model when there are 10, 20, and 40 nodes in the workloads. At modest scales, with fan-outs of 10 or even 20 nodes, there are substantial gains to be realized by avoiding bad nodes. However, there is less room for error here than in the sequential model: as the system scales up, other barriers present themselves, and avoiding nodes we classify as bad provides diminishing returns. Understanding this open question is an important challenge for us going forward.

4.2 System Design and Implementation

Bobtail needs to be a scalable system that makes accurate decisions in a timely fashion. While the node property remains stable in our five-week measurement, empirical evidence shows that the longer Bobtail runs, the more accurate its result can be. However, because launching an instance takes no more than a minute in EC2, we limit Bobtail to making a decision in under two minutes. Therefore, we need to strike a balance between accuracy and scalability.

A naive approach might be to simply conduct network measurements with every candidate. But however accurate it might be, such a design would not scale well to handle a large number of candidate instances in parallel: to do so in a short period of time would require sending a large amount of network traffic as quickly as possible to all candidates, and the synchronous nature of the measurement could cause severe network congestion or even TCP incast [77].

On the other hand, the most scalable approach involves conducting testing locally at the candidate instances, which does not rely on any resources outside the instance itself. Therefore, all operations can be done quickly and in parallel. This approach trades accuracy for scalability. Fortunately, Figures 4.1 and 4.2 show that Bobtail is allowed to make mistakes.

Based on our root cause analysis, such a method exists because the part of the long tail problem we focus on is *a property of nodes instead of the network*. Accordingly, if we know the workload patterns of the VMs co-located with the victim VM, we should be able to predict if the victim VM will have a bad latency distribution locally without any network measurement.

In order to achieve this, we must infer how often long scheduling delays happen to the victim VM. Because the long scheduling delays caused by the co-located CPU-intensive VMs are not unique to network packet processing and any interrupt-based events will suffer from the same problem, we can measure the frequency of large delays by measuring the time for the target VM to wake up from the `sleep` function call—the delay to process the timer interrupt is a proxy for delays in processing all hardware interrupts.

To verify this hypothesis, we repeat the five controlled experiments presented in the root cause analysis. But instead of running an RPC server in the victim VM and measuring the RTTs with another client, the victim VM runs a program that loops to sleep 1ms and measures the *wall time* for the `sleep` operation. Normally, the VM should be able to wake up after a little over 1ms, but co-located CPU-intensive VMs may prevent it from doing so, which results in large delays.

Figure 4.3 shows the number of times when the sleep time rises above 10ms in the five scenarios of the controlled experiments. As expected, when two or more VMs are CPU-intensive, the number of large delays experienced by the victim VM is *one to two orders of magnitude* above that experienced when zero or one VMs are CPU-intensive. Although the fraction of such large delays is small in all scenarios, the large difference in the raw counts forms a clear criterion for distinguishing bad nodes from good nodes. In addition, although it is not shown in the figure, we find that large delays with zero or one CPU-intensive VMs mostly appear for lengths of around 60ms or 90ms; these are caused by the

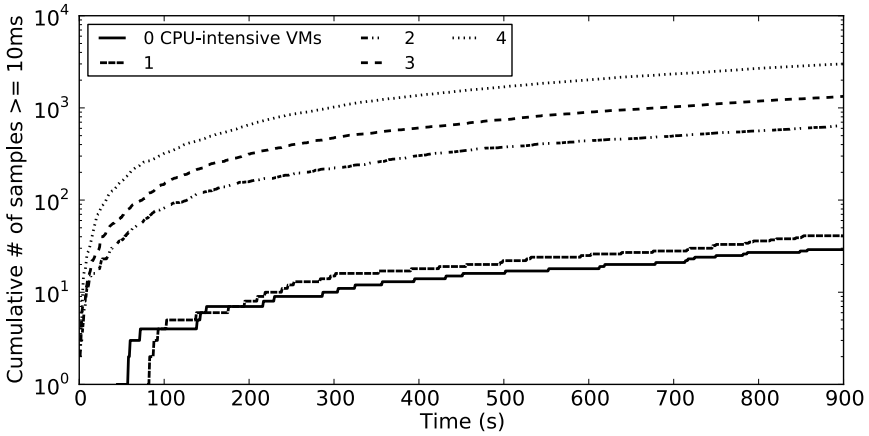


Figure 4.3: The number of large scheduling delays experienced by the victim VM in controlled experiments with an increasing number of VMs running CPU-intensive workloads. Such large delay counts form a clear criterion for distinguishing bad nodes from good nodes.

40% CPU cap on each latency-sensitive VM (i.e., when they are not allowed to use the CPU despite its availability). Delays experienced in other scenarios are more likely to be below 30ms, which is a result of latency-sensitive VMs preempting CPU-intensive VMs. This observation can serve as another clue for distinguishing the two cases.

Protocol 1 Instance Selection Algorithm

```
num_delay = 0
for i = 1 → M do
    sleep for S micro seconds
    if sleep time ≥ 10ms then
        num_delay++
    end if
end for
if num_delay ≤ LOW_MARK then
    return GOOD
end if
if num_delay ≤ HIGH_MARK then
    return MAY USE NETWORK TEST
end if
return BAD
```

Based on the results of our controlled experiments, we can design an instance selection algorithm to predict *locally* if a target VM will experience a large number of long scheduling delays. Algorithm 1 shows the pseudocode of our design. While the algorithm

itself is straightforward, the challenge is to find the right threshold in EC2 to distinguish the two cases (`LOW_MARK` and `HIGH_MARK`) and to draw an accurate conclusion as quickly as possible (loop size M).

Our current policy is to reduce false positives, because in the partition-aggregation pattern, reducing bad nodes is critical to scalability. The cost of such conservatism is that we may label good nodes as bad incorrectly, and as a result we must instantiate even more nodes to reach a desired number. To return N good nodes as requested by users, our system needs to launch $K * N$ instances, and then it needs to find the best N instances of that set with the lowest probability of producing long latency tails.

After Bobtail fulfills a user's request for N instances whose delays fall below `LOW_MARK`, we can apply the network-based latency testing to the leftover instances whose delays fall between `LOW_MARK` and `HIGH_MARK`; this costs the user nothing but provides further value using the instances that users already paid for by the hour. Many of these nodes are likely false negatives which, upon further inspection, can be approved and returned to the user. In this scenario, scalability is no longer a problem because we no longer need to make a decision within minutes. Aggregate network throughput for testing can be thus much reduced. With this optimization, we may achieve a much lower effective false negative rate, which will be discussed in the next subsection.

A remaining question is what happens if users run latency-sensitive workloads on the good instances Bobtail picked, but those VMs become bad after some length of time. In practice, because users are running network workloads on these VMs, they can tell if any good VM turns bad by inspecting their application logs without any extra monitoring effort. If it happens, users may use Bobtail to pick more good VMs to take the place of the bad ones. Fortunately, as indicated in Figure 3.3, our five-week measurement shows that such properties generally persist, so workload migration does not need to happen very frequently. In addition, Figures 4.1 and 4.2 also indicate that even if 20% of instances running latency-sensitive workloads are bad VMs, their impact on the latency distribution of sequential or partition-aggregation workloads is limited.

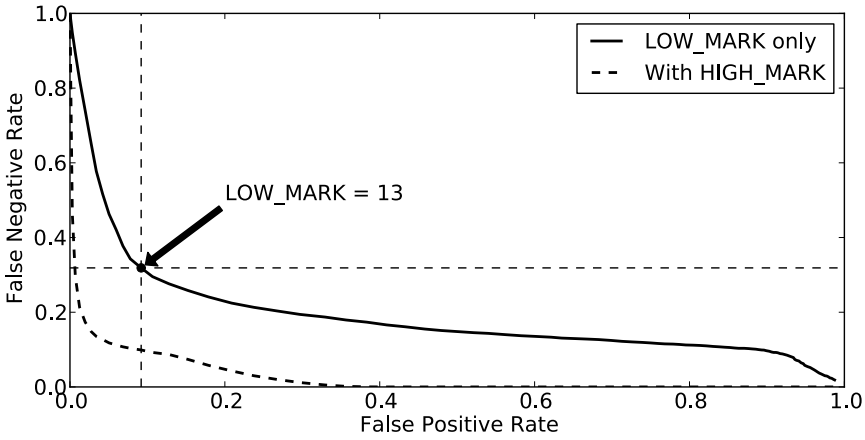


Figure 4.4: Trade-off between false positive and false negative rates of the instance selection algorithm. Our system can achieve a < 0.1 false positive rate while maintaining a false negative rate of around 0.3. With the help of network-based testing, the effective false negative rate can be reduced to below 0.1.

4.3 Parameterization

To implement Bobtail's algorithm, we need to define both its runtime (loop size M) and the thresholds for the `LOW_MARK` and `HIGH_MARK` parameters. Our design intends to limit testing time to under two minutes, so in our current implementation we set the loop size M to be $600K$ `sleep` operations, which translates to about 100 seconds on small instances in EC2—the worse the instance is, the longer it takes.

The remaining challenge we face is finding the right thresholds for our parameters (`LOW_MARK` and `HIGH_MARK`). To answer this inquiry, we launch 200 small instances from multiple availability zones (AZs) in EC2's US east region, and we run the selection algorithm for an hour on all the candidates. Meanwhile, we use the results of network-based measurements as the *ground truth* of whether the candidates are good or bad. Specifically, we consider the instances with 99.9th percentiles under 10ms for *all* micro benchmarks, which are discussed in § 4.4.1, as good nodes; all other nodes are considered bad.

Figure 4.4 shows the trade-off between the false positive and false negative rates by increasing `LOW_MARK` from 0 to 100. The turning point of the solid line appears when we set `LOW_MARK` around 13, which lets Bobtail achieve a < 0.1 false positive rate while maintaining a false negative rate of around 0.3—a good balance between false positive and false negative rates. Once `HIGH_MARK` is introduced (as five times `LOW_MARK`), the effective false

negative rate can be reduced to below 0.1, albeit with the help of network-based testing. We leave it as future work to study when we need to re-calibrate these parameters.

The above result reflects our principle of favoring a low false positive. Therefore, we need to use a relatively large K value in order to get N good nodes from $K*N$ candidates. Recall that our measured good node ratio for random instances directly returned by EC2 ranges from 0.4 to 0.7. Thus, as an estimation, with a 0.3 false negative rate and a 0.4 to 0.7 good node ratio for random instances from multiple data centers, we need $K*N*(1-0.3)*0.4 = N$ or $K \approx 3.6$ to retrieve the number of desired good nodes from one batch of candidates. However, due to the pervasiveness of bad instances in EC2, even if Bobtail makes no mistakes we still need a minimum of $K*N*0.4 = N$ or $K = 2.5$. If startup latency is the critical resource, rather than the fees paid to start new instances, one can increase this factor to improve response time.

4.4 Evaluation

In this subsection, we evaluate our system over two availability zones (AZs) in EC2's US east region. These two AZs always return some bad nodes. We compare the latency tails of instances both selected by our system and launched directly via the standard mechanism. We conduct this comparison using both micro benchmarks and models of sequential and partition-aggregation workloads.

In each trial, we compare 40 small instances launched directly by EC2 from one AZ to 40 small instances selected by our system from the same AZ. The comparison is done with a series of benchmarks; these small instances will run RPC servers for all benchmarks. To launch 40 good instances, we use $K = 4$ with 160 candidate instances. In addition, we launch four extra large instances for every 40 small instances to run RPC clients. We do this because, as discussed earlier, extra large instances do not experience the extra long tail problem; we therefore can blame the server instances for bad latency distributions.

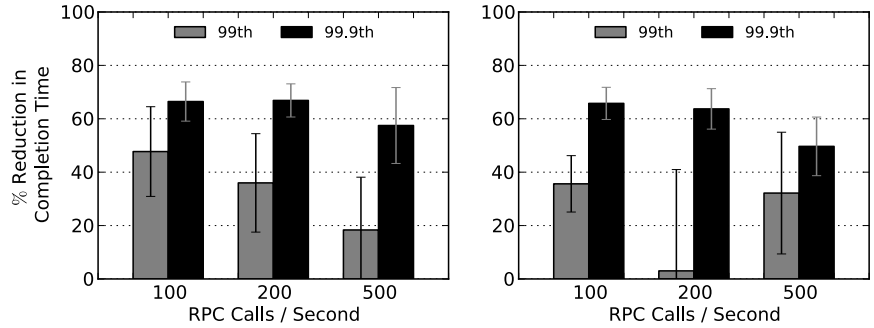


Figure 4.5: Reduction in flow tail completion time in micro benchmarks by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

4.4.1 Micro Benchmarks

Our traffic models for both micro benchmarks and sequential and partition-aggregation workloads have inter-arrival times of RPC calls forming a Poisson process. For micro benchmarks, we assign 10 small instance servers to each extra large client. The RPC call rates are set at 100, 200, and 500 calls/second. In each RPC call, the client sends an 8-byte request to the server, and the server responds with 2KB of random data. Meanwhile, both requests and responses are packaged with another 29-byte overhead. The 2KB message size was chosen because measurements taken in a dedicated data center indicate that most latency-sensitive flows are around 2KB in size [5]. Note that we do not generate artificial background traffic, because real background traffic already exists throughout EC2 where we evaluate Bobtail.

Figure 4.5 presents the reductions in completion times for three RPC request rates in micro benchmarks across two AZs. Bobtail reduces latency at the 99.9th percentile from 50% to 65%. In micro benchmark and subsequent evaluations, the mean of reduction percentages in flow completion is presented with a 90% confidence interval.

However, improvements at the 99th percentile are smaller with a higher variance. This is because, as shown in Figure 3.1, the 99th percentile RTTs within EC2 are not very bad to begin with ($\sim$2.5ms); therefore, Bobtail's improvement space is much smaller at the 99th percentile than at the 99.9th percentile. For the same reason, network congestion may have a large impact on the 99th percentile while having little impact on the 99.9th percentile in EC2. The outlier of 200 calls/second in the second AZ of Figure 4.5 is caused by one trial

in the experiment with 10 good small instances that exhibited abnormally large values at the 99th percentile.

4.4.2 Sequential Model

For sequential workloads, we apply the workload model to 20-node and 40-node client groups, in addition to the 10-node version shown in the micro benchmarks. In this case, the client sends the same request as before, but the servers reply with a message size randomly chosen from among 1KB, 2KB, and 4KB. For each workflow, instead of sending requests to all the servers, the client will randomly choose one server from the groups of sizes 10, 20, and 40. Then, it will send 10 synchronous RPC calls to the chosen server; the total time to complete all 10 RPC requests is then used as the workflow RTT. Because of this, the workflow rates for the sequential model are reduced to one tenth of the RPC request rates for micro benchmarks and become 10, 20, and 50 workflows per second.

Figure 4.6 shows our improvement under the sequential model with different numbers of RPC servers involved. Bobtail brings a 35% to 40% improvement to sequential workloads at the 99th percentile across all experiments, and it roughly translates to an 8ms reduction. The lengths of the confidence intervals grow as the number of server nodes increases; this is caused by a relatively smaller sample space. The similarity in the reduction of flow completion time with different numbers of server nodes shows that the tail performance of the sequential workflow model only depends on the ratio of bad nodes among all involved server nodes. Essentially, the sequential model demonstrates the average tail performance across all server nodes by randomly choosing one server node each time with equal probability at the client side.

Interestingly, and unlike in the micro benchmarks, improvement at the 99.9th percentile now becomes smaller and more variable. However, this phenomenon does match our simulation result shown in Figure 4.1 when discussing the potential benefits of using Bobtail.

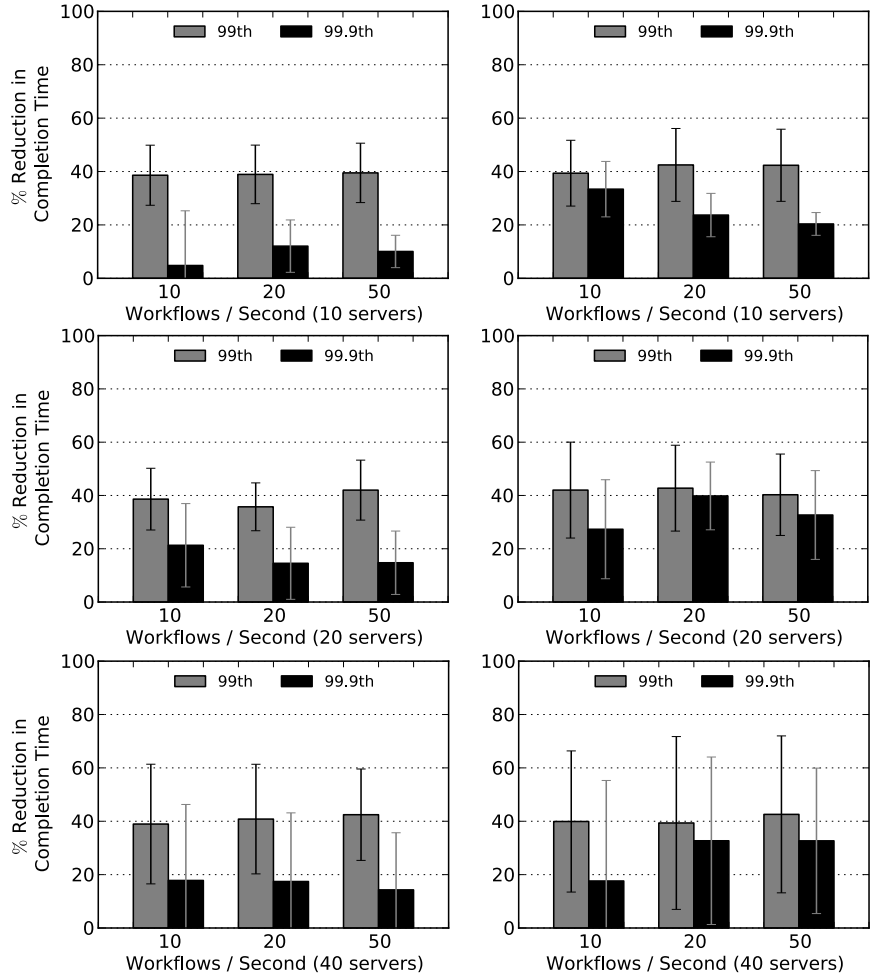


Figure 4.6: Reduction in flow tail completion time for sequential workflows by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

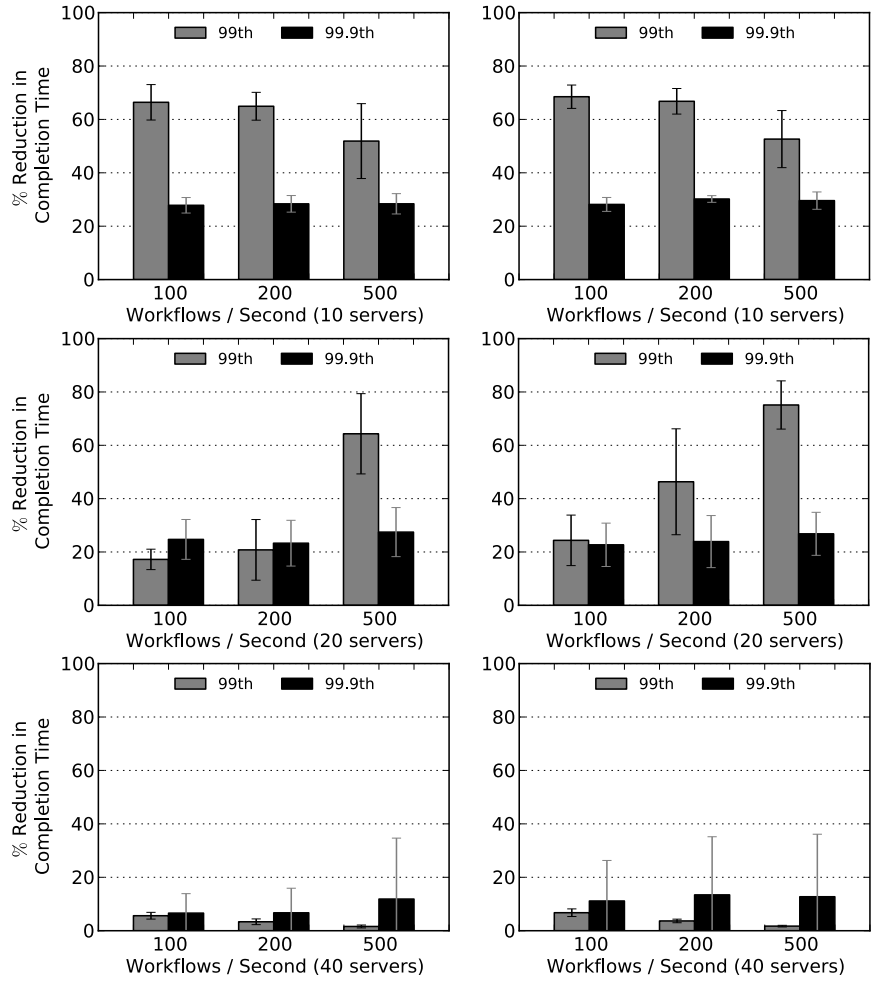


Figure 4.7: Reduction in flow tail completion time for partition-aggregation workflows by using Bobtail in two availability zones in EC2's US east region. The mean reduction time is presented with a 90% confidence interval.

4.4.3 Partition-Aggregation Model

For the partition-aggregation model, we use the same 10, 20, and 40-node groups to evaluate Bobtail. In this case, the client always sends requests to all servers in the group concurrently, and the workflow finishes once the slowest RPC response returns; servers always reply with 2KB of random data. In other words, the RTT of the slowest RPC call is effectively the RTT of the workflow. Meanwhile, we keep the same workflow request rate from the micro benchmarks.

Figure 4.7 shows improvement under the partition-aggregation model with different numbers of RPC servers involved. Bobtail brings improvement of 50% to 65% at the 99th

percentile with 10 servers. Similarly to the sequential workloads, the improvement at the 99.9th percentile is relatively small. In addition, as predicted by Figure 4.2, the reduction in tail completion time diminishes as the number of servers involved in the workload increases. To fully understand this phenomenon, we need to compare the behaviors of these two workload models.

For sequential workloads with random worker assignment, a small number of long-tail nodes have a modest impact. Intuitively, each such node has a $1/N$ chance of being selected for any work item and may (or may not) exhibit long tail behavior for that item. However, when this does happen, the impact is non-trivial, as the long delays consume the equivalent of many "regular" response times. So, one must minimize the pool of long-tail nodes in such architectures but needs not to avoid them entirely.

The situation is less pleasant for parallel, scatter-gather style workloads. In such workloads, long-tail nodes act as the barrier to scalability. Even a relatively low percentage of long-tail nodes will cause significant slowdowns overall, as each phase of the computation runs at the speed of the slowest worker. Reducing or even eliminating long-tail nodes removes this early barrier to scale. However, it is not a panacea. As the computation fans out to more nodes, other limiting factors come into play, reducing the effectiveness of further parallelization. We leave it as future work to study other factors that cause the latency tail problem with larger fan-out in cloud data centers.

4.5 Discussion

Emergent partitions A naive interpretation of Bobtail's design is that a given customer of EC2 simply seeks out those nodes which have at most one VM per CPU. If this were the case, deploying Bobtail widely would result in a race to the bottom. However, not all forms of sharing are bad. Co-locating multiple VMs running latency-sensitive workloads would not give rise to the scheduling anomaly at the root of our problem. Indeed, wide deployment of Bobtail for latency-sensitive jobs would lead to placements on nodes which are either under-subscribed or dominated by other latency-sensitive workloads. Surprisingly, this provides value to CPU-bound workloads as well. Latency-sensitive workloads

will cause frequent context switches and reductions in cache efficiency; both of these degrade CPU-bound workload performance. Therefore, as the usage of Bobtail increases in a cloud data center, we expect it will eventually result in emergent partitions: regions with mostly CPU-bound VMs and regions with mostly latency-sensitive VMs. This is also the key difference between Bobtail and other studies of placement gaming [62, 27] that exploit the performance variability in public clouds. However, to validate this hypothesis, we would need direct access to the low-level workload characterization of cloud data centers like EC2.

Alternative solutions Bobtail provides a guest-centric solution that cloud users can apply to avoid long latency tails without changing any of the underlying infrastructure. Alternatively, cloud providers can offer their solutions by modifying the cloud infrastructure and placement policy. For example, they can avoid allocating more than C VMs on a physical machine with C processors, at the cost of resource utilization. They can also overhaul their VM placement policy to allocate different types of VMs in different regions in the first place. In addition, new versions of the credit scheduler [88] may also help alleviate the problem. There two alternatives discussed in the rest of this dissertation; both approaches require to modify the cloud infrastructure below the virtualization semantic gap.

4.6 Summary

In this chapter, we present a guest-centric system, Bobtail, which mitigates the impact of performance interference on inter-VM network latency by proactively detecting and avoiding bad EC2 instances that exhibit the long tail latency problem without any changes to the cloud infrastructure. Bobtail leverages the key observation that when co-scheduled with CPU-bound VMs on the same physical machines, the interrupt processing in latency-sensitive VMs are delayed due to the contention of shared processors. Evaluations in two availability zones in EC2's US east region show that common communication patterns benefit from reductions of up to 40% in their 99.9th percentile response times.

CHAPTER 5

A Host-Centric Approach to Avoid Latency Traps

In this chapter, we take a host-centric approach to mitigate the impact of virtual machine performance interference on network latency. The solution described in this chapter holistically attacks all three latency problems characterized in the measurement study (Chapter 3): the VM scheduling delay, the host network queueing delay, and the switch queueing delay.

To address these problems using the same principle, we extend a classic scheduling policy—Shortest Remaining Time First—from the virtualization layer, through the host network stack, to the network switches. The intuition is to prioritize small tasks over large ones to reduce latency without undue harm to throughput. Unlike most existing solutions that require modification to the software stack controlled by guests, our solution strives to avoid trusting guest cooperation [93]. Compared to Bobtail (Chapter 4), this system is completely transparent to cloud customers. Experimental and simulation results show that our solution can reduce median latency of small flows by 40%, with improvements in the tail of almost 90%, while reducing throughput of large flows by less than 3%.

5.1 Related Work

Data center network latency Based on the design requirements, we classify existing latency reduction solutions as being *kernel-centric*, *application-centric*, or *operator-centric*. DCTCP [5] and HULL [6] are kernel-centric solutions because among other things, they both require modifying the operating system (OS) kernel to deploy new TCP congestion

control algorithms. On the other hand, applications can enjoy low-latency connections without any modification.

D^3 [82], D^2TCP [75], DeTail [95], PDQ [38], and pFabric [7] are application-centric solutions. While some of them also require modifying the OS kernel or switching fabrics, they share a common requirement. That is, applications must be modified to tag the packets they generate with *scheduling hints*, such as flow deadline or relative priority.

Operator-centric solutions require no changes to either OS kernel or applications, but they do require operators to change their application deployment. For example, Bobtail (Chapter 4), helps operator determine which virtual machines are suited to deploy latency-sensitive workloads without changing the applications or OS kernel themselves.

In a virtualized multi-tenant data center, all of these solutions are in fact *guest-centric*—they require changing the *guest* OS kernel, the *guest* applications themselves, or the way *guest* applications are deployed—none of which are controlled by cloud providers. In contrast, the solution described in this chapter is *host-centric*—it does not require or trust guest cooperation, and it only modifies the host infrastructure controlled by cloud providers.

EyeQ also adopts a host-centric design [42]. While it mainly focuses on bandwidth sharing in the cloud, like our work, EyeQ also discusses the trade-offs between throughput and latency. In comparison, our solution does not require feedback loops between hypervisors to coordinate, and it does not need explicit bandwidth headroom to reduce latency. Additionally, the bandwidth headroom used by EyeQ only solves one of the three latency problems addressed by our solution.

Virtual machine scheduling To improve Xen's I/O performance, new VM scheduling schemes, such as vSlicer [91], vBalance [18], and vTurbo [90], are proposed to improve the latency of interrupt handling by using a smaller time slice for CPU scheduling [91, 90] or by migrating interrupts to a running VM from a preempted one. However, unlike our host-centric design, these approaches either require modifications to the guest VMs [18, 90] or to trust the guests to specify their workload properties [91], neither of which are easily applicable to public clouds. Similarly, the soft real-time schedulers [53, 50, 89] designed to meet latency targets also require explicit guest cooperation. In addition, by monitoring

guest I/O events and giving preferential treatment to the I/O-bound VMs, their I/O performance can be improved without any guest changes [31, 44, 39]. In comparison, our design does not require such gray-box approaches to infer guest VM I/O activities, and thus avoids the complexity and overhead.

Meanwhile, using hardware-based solutions like Intel's Virtual Machine Device Queues (VMDq) [41], guest VMs can bypass the host operating system to handle packets directly and significantly improve their network I/O performance. The deployment of such hardware would eliminate the need to modify the host operating system to reduce the queueing delay in its software queues, which solves one of three latency problems discussed in this work. On the flip side, our modification to reduce the latency in the host network stack is software-only so that it is also applicable to the physical machines without the advanced hardware virtualization support.

Shortest remaining time first SRTF is a classic scheduling policy known for minimizing job queueing time [69] and widely used in system design. For networking systems, Guo *et al.* use two priority classes—small and large flows—to approximate the SRTF policy for Internet traffic; it obtains better response time without hurting long TCP connections [32]. Similarly, Harchol-Balter *et al.* change Web servers to schedule static responses based on their size using SRTF and demonstrate substantial reduction in mean response time with only negligible penalty to responses of large files [35]. In addition, pFabric uses SRTF to achieve near-optimal packet scheduling if applications can provide scheduling hints in their flows [7]. In this chapter, we apply SRTF holistically to three areas of virtualized data center infrastructure by following the same principle as these existing systems. Importantly, we approximate SRTF without requiring guest cooperation or explicit resource reservation.

5.2 Design and Implementation

Applying the prior solutions to reduce network latency in virtualized multi-tenant data centers would require guests to change their TCP implementation [5, 6], modify their applications to provide scheduling hints [7, 38, 75, 82, 95], or complicate their workload

deployment process (e.g., Bobtail in Chapter 4). Alternatively, network resources have to be reserved explicitly [42]. Meanwhile, even a combination of these approaches cannot solve all three latency problems discussed in Chapter 3.

Our solution has *none* of the preceding requirements and yet it holistically tackles the three latency traps—we only modify the host infrastructure in a way that is transparent to the guests, and there is no need to explicitly reserve network resources. The details of dealing with individual latency traps are discussed in the subsequent subsections, all of which follow the same principles:

Principle I: Not trusting guest VMs We do not rely on any technique that requires guest cooperation. The multi-tenant nature of public data centers implies that guest VMs competing for shared resources are greedy—they only seek to optimize the efficiency of their own resource usage. The prior solutions tailored for dedicated data centers [5, 6, 7, 38, 75, 82, 95] have proven very effective because the application stacks, operating systems, and hardware infrastructure are controlled by a single entity in such environments. In public data centers, however, trusting guests to cooperatively change the TCP implementation in the guest operating systems or provide scheduling hints would reduce the effectiveness or worse, the fairness of resource scheduling.

Principle II: Shortest remaining time first We leverage Shortest Remaining Time First (SRTF) to schedule bottleneck resources. SRTF is known for minimizing job queueing time [69]. By consistently applying SRTF from the virtualization layer, through the host network stack, to the data center network, we can eliminate the need of explicit resource reservation, but still significantly reduce network latency.

Principle III: No undue harm to throughput SRTF shortens latency at the cost of the throughput of large jobs; we seek to reduce the damage. Due to the fundamental trade-off between latency and throughput in system design, many performance problems are caused by the design choices made to trade one for another. Our solutions essentially revisit such choices in various layers of the cloud host infrastructure and make new trade-offs.

5.2.1 VM scheduling delay

§ 3.1 shows that VM scheduling can increase tail latency significantly. Such delay exists because Xen fails to allow latency-bound VMs to handle their pending interrupts soon enough. We claim that Xen's current VM scheduler does apply the SRTF principle, but it is applied *inadequately*. That is, there exists a mechanism in the scheduler to allow latency-sensitive VMs to preempt the CPU-bound VMs that mostly use 100% of their allocated CPU time, but it leaves a chance for the VMs that use less CPU time (e.g., 90%) to delay the interrupt handling of their latency-bound neighbors. Thus, our solution is to *apply SRTF in a broader setting* by allowing latency-sensitive VMs with pending interrupts to preempt *any* running VMs.

To understand the preceding argument, one needs to know Xen's VM scheduling algorithm. Credit Scheduler is currently Xen's default VM scheduler [88]. As the name implies, it works by distributing credits to virtual CPUs (VCPUs), which are the basic scheduling units. Each guest VM has at least one VCPU. By default, a VCPU receives up to 30ms CPU time worth of credits based on its relative weight. Credits are redistributed in 30ms intervals and burned when a VCPU is scheduled to use a physical CPU. A VCPU that uses up all its credits enters the `OVER` state, and a VCPU with credits remaining stays in the `UNDER` state. The scheduler always schedules `UNDER`s before any `OVER`s. Importantly, if a VCPU waken up by a pending interrupt has credits remaining, it enters the `BOOST` state and is scheduled before any `UNDER`s.

The `BOOST` state is the mechanism designed to approximate SRTF: A VCPU that only spends brief moments handling I/O events is considered a small job; hence it is favored by the scheduler to preempt CPU-bound jobs as it stays in the `BOOST` state. Unfortunately, latency-bound VMs may still suffer from large tail latency despite the `BOOST` mechanism and this is a known limitation [25].

By analyzing Xen's source code, we find that the credit scheduler is still biased far towards throughput. The `BOOST` mechanism only prioritizes VMs over others in `UNDER` or `OVER` states; `BOOST`ed VMs cannot preempt each other, and they are round-robin scheduled. Thus, if a VM exhausts its credits quickly, it stays in the `OVER` state most of time, and

BOOSTed VMs can almost always preempt it to process their interrupts immediately. However, if a VM sleeps to accumulate credits for a while and then wakes up on an interrupt, it can monopolize the physical CPU by staying BOOSTed until its credits are exhausted, which implies a delay of 30ms or even longer (if more than one VM decide to do that) to neighboring VMs that are truly latency-sensitive.

This problem can be solved by *applying SRTF in a broader setting*. We deploy a more aggressive VM scheduling policy to allow BOOSTed VMs to preempt each other. This change is only made to the hypervisor (one line in the source code!) and thus transparent to guest VMs. As a result, true latency-bound VMs can now handle I/O events more promptly; the cost is CPU throughput because job interruption may become more frequent under the new policy. However, such preemption cannot happen arbitrarily: Xen has a `rate limit` mechanism that maintains overall system throughput by preventing preemption when the running VM has run for less than 1ms in its default setting.

5.2.2 Host network queueing delay

§ 3.3.3 shows that guest VMs doing bulk transferring can increase their neighbor's host network queueing delay by four to six times, even when BQL and CoDel are both enabled. The root cause is the contention of the queueing space in the host network stack—network requests from bandwidth-bound guest VMs are often *too large and hard to be preempted* in Linux kernels' and NICs' transmission queues. Thus, our solution is to *break large jobs into smaller ones* to allow CoDel to conduct fine-grained packet scheduling.

To understand the above argument, we need to explain Xen's approach to manage network resources for guest VMs. Xen uses a split driver model for both network and disk I/O. That is, for each device, a guest VM has a virtual device driver called `frontend`, and the dom0 has a corresponding virtual device driver called `backend`; these two communicate by memory copy. A packet sent out by guest VMs first goes to the `frontend`, which copies the packet to the `backend`. The job for the `backend` is to communicate with the real network device driver on dom0 to send out packets on the wire. To do so, Xen leverages the existing routing and bridging infrastructure in the Linux kernel by treating the virtual `backend` as a

real network interface on the host so that every packet received by a `backend` will be routed or bridged to the physical NICs. Packet reception simply reverses this routing/bridging and memory copy process.

The overhead of packet copy between guest and host may be prohibitive during bulk transferring. Thus, Xen allows guest VMs to consolidate outgoing packets to up to 64KB regardless of NICs' MTU (e.g., 1,500 bytes for Ethernet). Unfortunately, this optimization also exacerbates host network queueing delay. Recall that the key to the success of BQL and CoDel is the fine-grained control of transmission queue length. By allowing bursts of 64KB packets, there are often *large jobs blocking the head of the line of host transmission queues*. Thus, our solution is to *segment large packets into smaller ones* so that CoDel would allow packets of latency-sensitive flows (small jobs) to preempt large bursts.

The question is *when* to segment large packets. Recall that to use BQL and CoDel effectively, best practice suggests turning off hardware segmentation offloading to segment large packets in software [86]. Unfortunately, it only works for packets generated by dom0 itself; traffic from guest VMs is routed or bridged directly, without segmentation, before reaching the NICs. Thus, a straightforward solution is simply to disallow guest VMs to send large packets and force them to segment in software before packets are copied to `backend`. However, according to Principle I, we cannot rely on guests to do so cooperatively, and it consumes guests' CPU cycles. Alternatively, Xen's `backend` can announce to the guest VMs that hardware segmentation offload is not supported; then guests have to segment the packets before copying them. While this approach achieves the goal without explicit guest cooperation, it disables Xen's memory copy optimization completely.

Thus, the key to a practical solution is to delay the software segmentation as much as possible—from guest kernel to host device driver, earlier software segmentation implies higher CPU overhead. Our solution is to segment large packets *right before they join the software scheduling queue managed by CoDel in the host network stack*. In order to give CoDel fine-grained jobs to schedule, this is the latest point possible.

5.2.3 Switch queueing delay

§ 3.3.2 shows that switch queueing delay can increase median latency by 2X and cause a 10X increase at the tail. This problem is not unique to public data centers but also exists in dedicated data centers. However, the added layer of virtualization exacerbates the problem and renders the solutions that require kernel or application cooperation impractical because these software stacks are now controlled by the guests instead of cloud providers.

Our host-centric solution approximates SRTF by letting switches favor small flows when scheduling packets on egress queues. The key insight here is that we can infer small flows like CPU schedulers infer short tasks—based on their resource usage (e.g., bandwidth). Importantly, we *infer flow size in the host* before any packets are sent out. This design also avoids explicit resource reservation required by alternative host-centric solutions [42]. In order to realize this design, we need to answer a few questions: a) How to define a flow? b) How to define flow size? c) How to classify flows based on their size?

To define what a flow is for the purpose of network resource scheduling, we first need to understand what the bottleneck resource is. Because switch queueing delay on a link is proportional to the occupancy of the switch egress queue, any packet would occupy the queueing resource. Thus, we define a flow as the collection of *any* IP packets from a source VM to a destination VM. We ignore the natural boundary of TCP or UDP connections because, from a switch's perspective, one TCP connection with N packets occupies the same amount of resources as two TCP connections with N/2 packets each for the same source-destination VM pair.

By ignoring the boundary of TCP or UDP connections, the size of a flow in our definition can be arbitrarily large. Therefore, we adopt a message semantic by treating a query or a response as the basic communication unit and define flow size as the *instant size of a message* the flow contains instead of the absolute size of the flow itself. In reality, it is also difficult to define message boundaries. Thus, we measure flow by *rate* as an approximation of the message semantic. That is, if a flow uses a small fraction of the available link capacity in small bursts, it is sending small messages and thus treated as a small flow. In the context of SRTF policy, it states that a flow at a low sending rate behaves just like a short

job with respect to the usage of switch queueing resources.

Based on the preceding definitions, we classify flows into two classes, small and large, as is done by systems that apply SRTF to Internet traffic [32] and Web requests [35]. Bottleneck links can now service small flows with a higher priority than the large ones. As in the prior work, lower latency of small flows is achieved at the cost of the throughput of large flows. To ensure scalability and stay transparent to the guests, the classification is done in the host of the source VM of the target flow. Each packet then carries a tag that specifies its flow classification for bottleneck switches to schedule.

One important advantage of this rate-based scheme is that it can avoid starving large TCP connections. Imagine if we use TCP connection boundaries and define flow size as the number of packets each TCP connection contains. A large TCP connection may starve if the capacity of a bottleneck link is mostly utilized by high-priority small flows. This is because a large TCP connection lowers its sending rate for packet drops to avoid congestion, but small flows often send all packets before reaching the congestion avoidance stage. If we classify TCP connections by their absolute size, the large ones are always tagged as low priority and serviced after the small ones, regardless of their instant bandwidth usage. However, if classified by rate, a low priority flow may eventually behave just like a high priority one, by dropping its sending rate below a threshold. It will then get its fair share on the bottleneck link.

To implement this policy, we need two components. First, we build a monitoring and tagging module in the host that sets priority on outgoing packets without guest intervention. Small flows are tagged as high-priority on packet headers to receive preferential treatment on switches. Second, we need switches that support basic priority queueing; but instead of prioritizing by application type (port), they can just leverage the tags on packet headers. It is common for data center grade switches to support up to 10 priority classes [82], while we only need two. Moreover, if more advanced switching fabrics like pFabric[7] become available, it is straightforward to expand our solution to include fine-grained classification for more effective packet scheduling.

In our current implementation, the monitoring and tagging module is built into Xen's network `backend` running as a kernel thread in dom0. This is a natural place to implement

such functionality because `backend` copies every packet from guest VMs, which already uses a considerable amount of CPU cycles (e.g., 20%) at peak load; the overhead of monitoring and tagging is therefore negligible. In addition, because the overhead of monitoring and tagging is proportional to packet count instead of packet size, retaining guest VMs' ability to consolidate small packets to up to 64KB further improves this module's performance.

Finally, the flow classification algorithm is implemented using a standard token bucket meter. We assign a token bucket to each new flow with a `rate` and a `burst` parameter. `Rate` determines how fast tokens (in bytes) are added to the bucket, and `burst` determines how many tokens can be kept in the bucket unused. The token bucket meters a new packet by checking if there are enough tokens to match its size, and it consumes tokens accordingly. If there are enough tokens, the current sending rate of the flow is considered conformant, and the packet is tagged with high priority. Otherwise, it is classified as bandwidth-bound and serviced on a best effort basis. Low priority flows are re-checked periodically in case their sending rates drop. Similar to [32], flows are garbage-collected if they stay inactive for long enough. A flow may accumulate tokens to send a large burst, so `burst` limits the maximum tokens that a flow can consume in one shot.

5.2.4 Putting it all together

Our design revisits the trade-offs between throughput and latency. For VM scheduling delay, we apply a more aggressive VM preemption policy to Xen's VM scheduler at the cost of the efficiency of CPU-bound tasks. For host network queueing delay, we segment large packets from guests earlier in the stack for fine-grained scheduling with higher host CPU usage. For switch queueing delay, we give preferential treatment to the small flows at a low sending rate on switches with the loss of throughput for large flows. The performance trade-offs of this design are evaluated in § 5.3, and the possibility of gaming these new policies is discussed in § 5.4.

In our current implementation, we change a single line in the credit scheduler of Xen 4.2.1 to enable the new scheduling policy. We also modify the CoDel kernel module in

Linux 3.6.6 with about 20 lines to segment large packets in the host. Finally, we augment the Xen's network `backend` with about 200 lines of changes to do flow monitoring and tagging.

5.3 Evaluation

We use both a small testbed and an ns-3-based [59] simulation to do our evaluation. The testbed consists of five four-core physical machines running Linux 3.6.6 and Xen 4.2.1. They are connected to a Cisco Catalyst 2970 switch, which supports priority queueing (QoS). All NICs are 1Gbps with one `tx` ring. We start with a setup that includes all three latency traps to provide a big picture of the improvement our holistic solution brings before evaluating them individually. In addition to evaluating all three problems on the testbed, we use ns-3 to demonstrate the trade-off at scale for switch queueing delay. The other two problems are local to individual hosts, so no large-scale simulation is needed.

The workload for testbed evaluation models the query-response pattern. As in § 3.3.1, client VMs measure the round-trip times of each query-response pair as flow completion times (FCTs). Because the size of a response may range from a few kilobytes to tens of megabytes, FCT serves as a metric for both latency-sensitive traffic and bandwidth-bound traffic. This setup is similar to prior studies [6, 95] for dedicated data centers. In addition, `iperf` is used when we only need to saturate a bottleneck link without measuring FCT.

The parameters for flow classification include `rate` and `burst` for the token buckets meters, and the timers for cleaning inactive flows and re-checking low priority flows. We set the timers to be 10s and 100ms, respectively, and use 30KB for `burst` and 1% of the link capacity or 10Mbps for flow `rate`. Optimal values would depend on the traffic characteristics of the target environment, so we use the preceding values as our best guest for testbed evaluation and explore the parameter sensitivity using simulation. Meanwhile, measurement studies for data center traffic may help narrow down parameter ranges. For example, Benson *et al.* show that 80% of flows are under 10KB in various types of data centers [13]. Others have reported that flows under 1MB may be time sensitive, and such flows often follow a query-response pattern [5, 82]. Thus, setting `burst` anywhere between

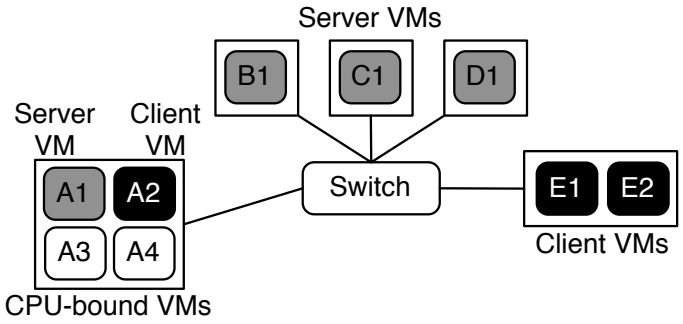


Figure 5.1: The testbed experiment setup.

10KB and 1MB would be a reasonable choice.

5.3.1 Impact of the approach

We first evaluate the overall benefits of our host-centric solution on the testbed by constructing a scenario with all three latency traps enabled. This scenario includes the *typical* network resource contention that could happen in virtualized multi-tenant data centers. We compare three cases: the ideal case without any contention, the fully contended case running unpatched Linux and Xen, and the patched case with our new policies. In the unpatched case, BQL and CoDel are enabled to represent the state-of-the-art prior to our solution.

Figure 5.1 shows the experiment setup. To create contention, we have physical machine E running two VMs, E1 and E2, that serve as clients to send small queries by following a Poisson process and measure FCT. Physical machine A runs four VMs—A1 through A4—with A1 serves small responses to E1, A2 sends bulk traffic using `iperf` to B1 on machine B, and the others run CPU-bound tasks. The query-response flows between E1 and A1 will suffer *VM scheduling delay* caused by A3 and A4, and *host network queueing delay* caused by A2. Moreover, we use physical machine C and D, running VM C1 and D1 respectively, to respond to E2's queries for large flows and congest E's access link, and the responses sent to E1 will suffer *switch queueing delay*. The small query flows generated by E1 expect 2KB responses with 5ms mean inter-arrival time. Queries generated by E2 expect 10MB large responses with 210ms mean inter-arrival time, so E's access link is 40% utilized on

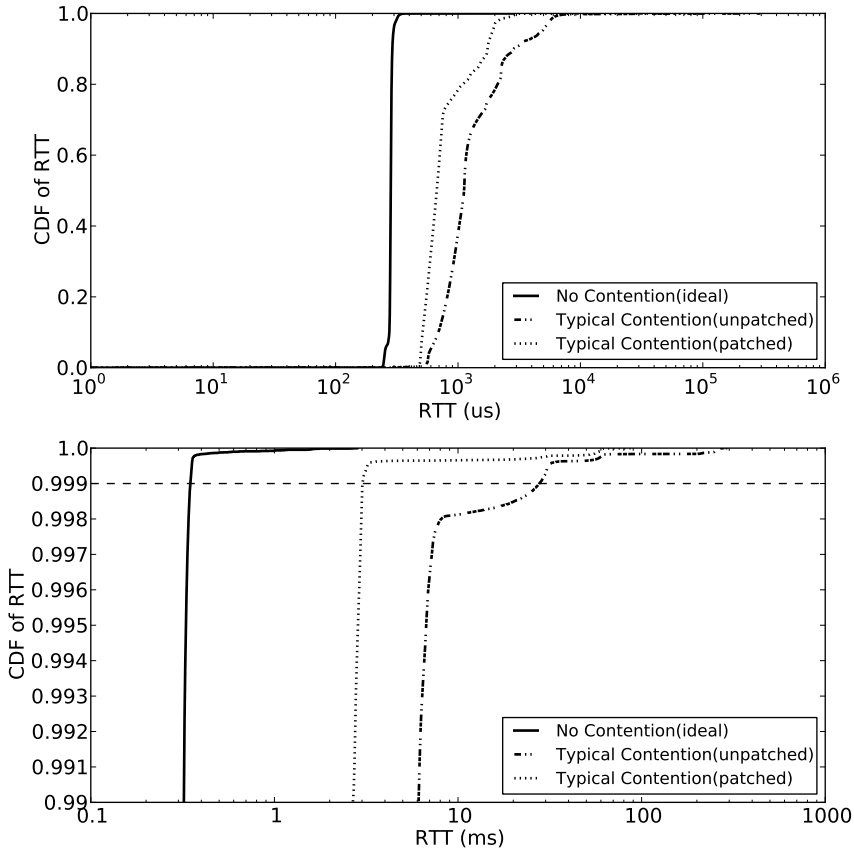


Figure 5.2: A comparison of FCT distribution for three cases: an ideal case without contention, the default unpatched case under typical contention, and the case patched with our solution under the same contention.

average, which is more likely than full saturation [6].

Figure 5.2 depicts the results. Our solution achieves about 40% reduction in mean latency, over 56% for the 99th percentile, and almost 90% for the 99.9th percentile. While the alleviation of host and switch queueing delay has a large impact on the 99th and lower percentiles, the improvement for the 99.9th percentile is mostly attributed to the change in the VM scheduler. However, compared to the baseline with no contention, there is still room for improvement, which we discuss when evaluating individual components. On the other hand, the average throughput loss for large flows is less than 3%. The impact on CPU-bound tasks and host CPU usage is covered in detail in subsequent subsections.

Scenarios	99.9th Latency	Avg. CPU Throughput
Xen Default	29.64ms	224.29 ops/s
New Policy	1.35ms	215.80 ops/s

Table 5.1: The trade-off between network tail latency and CPU throughput measured by memory scans per second.

5.3.2 VM scheduling delay

To evaluate the new VM scheduling policy, we use E1 to query A1 for small responses and keep A3 running a CPU-bound workload to delay A1's interrupt handling. In order to cause scheduling delay, A1 and A3 need to share a CPU core.Thus, we pin both VMs onto one CPU core and allocate 50% CPU cycles to each VM. Note that the delay caused by VM scheduling happens when a query flow from E1 arrives at A1 and before A1 is allowed to process its (virtual) network interrupt.

Now the question is what CPU-bound workload we use to demonstrate the trade-off between network tail latency and CPU throughput. As explained in § 5.2.1, a workload that always uses 100% CPU time cannot do the job. Instead, we need a workload that follows the pattern of yielding CPU briefly between CPU-hogging operations in order to accumulate credits; then it can get `BOOST`ed and delay its neighbors' interrupt handling. For example, a workload can sleep 1ms for every 100 CPU-bound operations. Note that this pattern is not uncommon for real-world workloads, which often wait for I/O between CPU-bound operations.

The next question is what operations we use to hog the CPU and exhibit slowdown when preempted. We cannot use simple instructions like adding integers because they are not very sensitive to VM preemption: If such operations are interrupted by VM preemption and then resumed, the penalty to their running time is negligible. We instead let the workload burn CPU cycles by scanning a CPU-L2-cache-sized memory block repeatedly. Such operations are sensitive to CPU cache usage because if certain memory content is cached, the read operations finish much faster than the ones fetching from memory directly, and the CPU cache usage is in turn sensitive to VM preemption and context switching [66, 76].

In this experiment, trade-offs between tail latency and CPU throughput are demonstrated for Xen's default scheduling policy and our new policy. To do so, we fix both the

number of network requests sent to A1 and the number of memory scan operations conducted on A3 so that each workload contains the same amount of work in both scenarios. The number of memory scan operations per second measured on A3 is used to quantify the throughput for the CPU-bound job.

Table 5.1 summarizes the results. Our new policy reduces network latency at the 99.9th percentile by 95% at the cost of 3.8% reduction in CPU throughput. In addition to the synthetic workload, we also tested with two SPEC CPU2006 [36] benchmarks, `bzip2` and `mcf`. The difference of their running time under different scheduling policies are less than 0.1%.

The new policy hurts CPU throughput because, while the workloads generate the same number of I/O interrupts in both scenarios, the CPU-bound job is more likely to be preempted in the middle of a scan operation under the new policy, which incurs the overhead of cache eviction and context switching. In comparison, Xen's default policy is more likely to allow I/O interrupts to be handled when the CPU-bound job is in the sleep phase.

To understand why our new policy only introduces low overhead, we record VCPU state-changing events in the VM scheduler since the loss in throughput is mostly caused by the extra VM preemption. We find that our new policy does increase the number of VM preemption, but the difference is only 2%. This is because the original `BOOST` policy already allows frequent preemption, and our change only affects the cases that correspond to the tail latency (99.9th or higher).

Further improvement is possible. Xen has the default policy that prevents VM preemption when the running VM has run for less than 1ms. If we set it to be the minimum 0.1ms, the 99.9th percentile latency can be reduced even further. However, such change has been shown to cause significant performance overhead to certain CPU-bound benchmarks [88] and may compromise fairness.

5.3.3 Host network queueing delay

For host network queueing delay, our setup is similar to the testbed experiment in § 5.2.2. Specifically, we use A1 to `ping` E1 once every 10ms for round-trip time (RTT)

Scenarios	50th	99th	99.9th
Congestion Free	0.217	0.235	0.250
Congestion Enabled	16.83	21.57	21.73
Congestion Managed	0.808	1.21	1.26
Our System	0.423	0.609	0.650

Table 5.2: The distribution of RTTs in millisecond. Our solution delivers 50% reduction in host network queueing delay in addition to that achieved by BQL and CoDel.

measurements and use A2 to saturate B1's access link with `iperf`, and we measure bandwidth. Because E1 and B1 use different access links, there is no switch queueing delay in this experiment. Throughout the experiment, hardware segmentation offload is turned off in order to use BQL and CoDel effectively [86].

Table 5.2 lists the results for the impact on `ping` RTTs. Compared to the case that applies BQL and CoDel unmodified (Congestion Managed), our solution can yield an additional 50% improvement at both the body and tail of the distribution because CoDel is more effective in scheduling finer-grained packets (small jobs). Meanwhile, the bandwidth loss for breaking down large packets early is negligible.

However, we do trade CPU usage for lower latency. When B1's 1Gbps access link is saturated, our current implementation increases the usage of one CPU core by up to 12% in the host due to earlier software segmentation, and that amount is negligible for guest VMs. Compared to the alternative that forces guest VMs to segment large packets without consolidation, hosts' CPU usage would increase by up to 30% from the overhead of copying small packets from guests, and that for the guest VMs would increase from 13% to 27% for its own software segmentation. Thus, our solution is not only transparent to users, but has less CPU overhead for both host and guest. This is because our choice of software segmentation is early enough to achieve low latency but late enough to avoid excessive CPU overhead.

Again there is still room for improvement. We speculate that the source of the remaining delay is from NICs' transmission queues. Recall that we set that queue to hold up to 37,500 bytes of packets, which translates to a 300μs delay on a 1Gbps link at maximum. To completely eliminate this delay, if necessary, we need higher-end NICs that also support priority queueing. Then, the NICs' drivers can take advantage of the flow tagging we

		2KB FCT (ms)					10MB FCT (ms)				
		avg.	50th	90th	99th	99.9th	avg.	50th	90th	99th	99.9th
20% Load	QoS Disabled	0.447	0.300	0.450	3.630	5.278	107.402	91.144	158.530	252.537	332.611
	QoS Enabled	0.298	0.298	0.319	0.357	0.499	109.809	91.308	159.768	314.574	431.564
40% Load	QoS Disabled	0.779	0.304	1.845	5.155	5.437	134.294	99.497	217.031	392.592	539.759
	QoS Enabled	0.305	0.301	0.333	0.441	0.533	138.002	104.164	222.907	419.591	639.909
60% Load	QoS Disabled	1.428	0.408	4.618	5.293	5.497	182.512	149.366	325.482	644.225	907.877
	QoS Enabled	0.319	0.305	0.363	0.514	0.551	187.316	153.745	338.976	646.169	908.825

Table 5.3: The results for tackling switch queueing delay with flow tagging and QoS support on the switch. Both the latency-sensitive flows and bandwidth-bound flows are measured by their FCT.

assigned to the latency-sensitive flows and send them before any packets of large flows.

5.3.4 Switch queueing delay

To evaluate the flow monitoring and tagging module, we leave VMs A2 through A4 idle. For the rest of the VMs, E1 queries A1 and B1 in parallel for small flows, and E2 queries C1 an D1 for large flows to congest the access link shared with E1 and cause switch queueing delay.

Similar to the "Dynamic Flow Experiments" used by Alizadeh *et al.* [6], we set the utilization of the access links to be 20%, 40%, and 60% in three scenarios, respectively, instead of fully saturating them. To do so, VM E2 runs a client that requests large response flows from C1 and D1 in parallel; it varies query rate to control link utilization. For example, this experiment uses 10MB flows as large responses, so we approximate 40% link utilization by using a Poisson process in E2 to send 5 queries per second on average. The case with the switch QoS support enabled is compared against the one without QoS support; QoS enabled switches can prioritize packets tagged as belonging to small flows.

Table 5.3 summarizes the results. As expected, when QoS support on the switch is enabled to recognize our tags, all small flows enjoy a low latency with an order of magnitude improvement at both the 99th and 99.9th percentiles. As the link utilization increases, their FCT increases only modestly. On the other hand, the *average* throughput loss for large flows is less than 3% In fact, the average throughput is not expected to experience a significant loss under the SRTF policy if the flow size follows a heavy-tail distribution [35]. This is because under such distribution, bottleneck links are not monopolized by high priority small flows and they have spare capacity most of time to service large flows with

modest delay. Benson *et al.* show that the flow size in various cloud data centers does fit heavy-tailed distributions [13].

Meanwhile, the increase in FCT for 10MB flows at the 99th and 99.9th percentiles can be as large as 30%, which is comparable to HULL [6]. This is expected because for 1% or 0.1% of these low-priority flows, there could be high-priority packets that happen to come as a stream longer than average. Because a switch egress queue has to service the high-priority packets first, before any best-effort packets, a few unlucky large flows have to be stuck in the queue substantially longer than average. In fact, the larger the flows are, the more likely they are affected by a continuous stream of high priority packets. On the other hand, larger flows are also less sensitive to queueing delay, and tail FCT only has marginal impact on their average throughput.

For testbed experiments, because the size of both small and large flows is fixed, all flows are correctly tagged. To demonstrate the sensitivity of the parameters, we need to test the solution against dynamically generated flows that follow a heavy-tail distribution, which is discussed in the next subsection.

5.3.5 Large-scale simulation

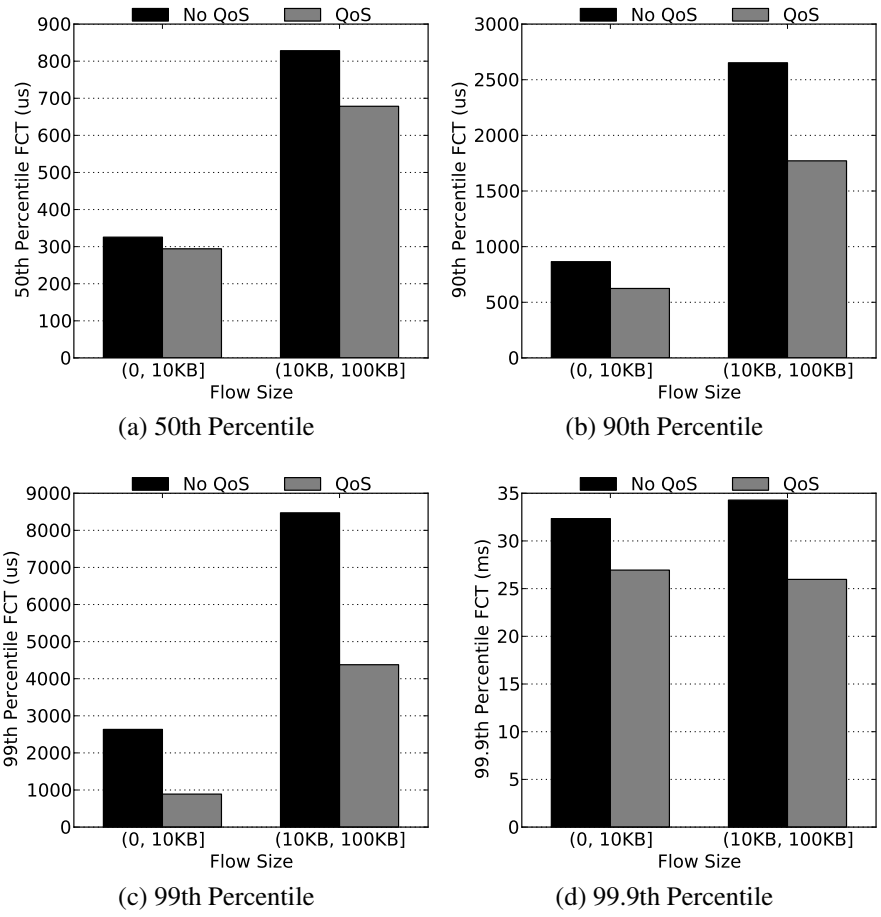
To explore the scalability and parameter sensitivity of our solution for switch queueing delay, we use ns-3 to simulate a multi-switch data center network. Because only switch queueing delay is evaluated here, we install multiple applications on each simulated host to mimic multiple VMs. All applications have their own sockets that may transmit or receive traffic simultaneously to cause congestion. The flow monitoring and tagging module also uses a 10Mbps `rate` 30KB `burst` token bucket meter per flow to begin with, and we vary them in the parameter sensitivity analysis.

Our simulated network follows the fat-tree topology [4]. There are 128 simulated hosts divided into four pods. In each pod, four 8-port edge switches are connected to the hosts and to four aggregation switches. In the core layer, 8 switches connect the four pods together. The links between hosts and edge switches are 1Gbps, and those between switches are 10Gbps, so that there is no over-subscription in the core network. We set each edge switch

to incur a 25μs delay and for upper-level switches, a 14.2μs delay. Thus, the intra-switch RTT is 220μs, the intra-pod RTT is 276.7μs, and the inter-pod RTT is 305.2μs. The routing table of the network is generated statically during simulation setup, and we use flow hashing to do load balancing.

The workload we use in the simulation is the same as in a prior study [6]. Specifically, each host opens a permanent TCP connection to all other hosts and chooses destinations at random to send flows to. Therefore, the FCT here is the time to finish a one-way flow instead of a query-response pair. The flows are generated by following a Poisson process with 2ms inter-arrival time on average. In addition, the flow size is generated using a Pareto random variable with the same parameters as in [6]: 1.05 for the shape parameter and 100KB mean flow size. As result, the utilization of access links is approximately 40%. This flow size distribution is picked to match the heavy-tailed workload found in real networks: most flows are small (e.g., less than 100KB), while most bytes are from large flows (e.g., 10MB and larger).

Figure 5.3 shows the FCT for small flows of range (0, 10KB] or (10KB, 100KB]; the flow size categorization is also based on [6]. We can observe improvement for both types of small flows at all percentiles, and the improvement ranges from 10% to 66%. However, unlike the testbed results, the FCTs with QoS enabled in the simulation result are not consistently low. While the 50th, 90th, and 99th percentile values are all comparable to the testbed results, the 99.9th percentile is an order of magnitude higher, and we only achieve 17% and 24% improvement for both types of small flows, respectively. This is an expected result because the flow sizes vary in the simulation and the flow tagging module may not be able to tag every flow correctly. More importantly, our tagging algorithm has an implicit assumption that a small flow stays small most of time, and a large flow also stays large most of time, which is not unreasonable for real world workload patterns. However, the simulated workload maintains permanent TCP connections for every pair of applications (VMs), and the flow size for each connection alone follows the Pareto distribution. As a result, a connection may transmit a random mix of flows with different sizes so that there is no correct label to be assigned to any connection. The flow tagging module is therefore more prone to making mistakes. As discussed in § 5.4, evaluating with real world flow



(a) 50th Percentile

(b) 90th Percentile

(c) 99th Percentile

(d) 99.9th Percentile

Figure 5.3: The 50th, 90th, 99th, 99.9th percentile FCT for small flows of range (0, 10KB] and (10KB, 100KB].

Flow Size	No QoS (ms)	QoS (ms)
(100KB, 10MB]	11.878	10.608
(10MB, ∞)	669.618	658.404

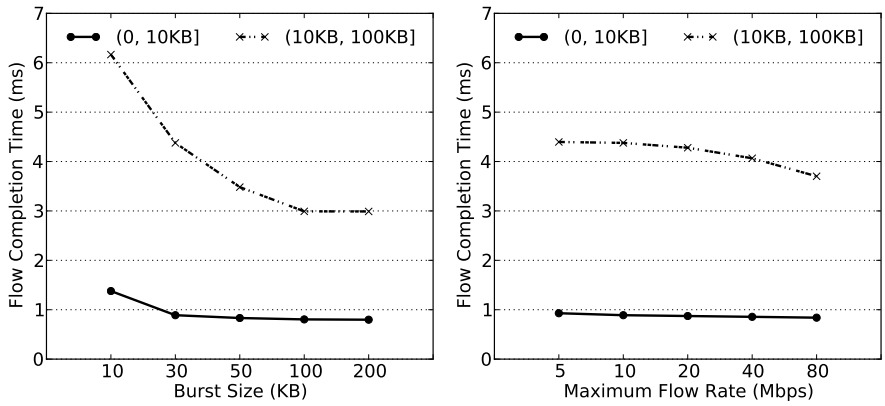
Table 5.4: The average FCTs for large flows.

traces would give us a better understanding of the problem.

Table 5.4 compares the average FCTs for large flows. To some degree, the FCTs are similar regardless of whether QoS is enabled or not, which is expected because when all the access links are 40% utilized and the core network is not over-subscribed, large flows are not expected to be starved by small flows. On the other hand, however, the numbers with QoS enabled are slightly smaller than that without QoS. This is the *opposite* of the testbed results. There are at least two possible explanations. First, our implementation of priority queueing and buffer management on simulated switches is very primitive compared to that in real Cisco switches. Thus, certain performance-related factors associated with priority queueing may be missing in the simulation. Secondly, with QoS enabled, small and large flows are classified into two separate queues on simulated switches, thus bandwidth-bound flows are protected from packet drops caused by bursts of small flows and hence operate in a more stable environment. A similar speculation is made in the study that applies SRTF to Internet traffic [32]. Harchol-Balter *et al.* also find that for Web requests of static files using a two-class SRTF scheduling can benefit *all* requests [35].

Finally, we evaluate the sensitivity of `burst` and `rate` for flow monitoring and tagging. The choices for these parameters depend on the flow characteristics of the target data center environment. According to recent data center measurements [13, 5, 82], any value in the range of [10KB, 1MB] may be reasonable for `burst` as the size of a single message. Once `burst` is determined, `rate` determines how many query-response pairs can be exchanged per second before a flow is classified as low priority. To demonstrate the sensitivity, we first fix `burst` to be 10Mbps and vary `burst` to be {10, 30, 50, 100, 200}KB. Then, we fix `burst` to be 30KB and use {5, 10, 20, 40, 80}Mbps as `rate` values.

Figure 5.4 shows the 99th percentile FCT for small flows with varying burst sizes or flow rates. Overall, the FCT improves as burst size increases until 100KB. It is the mean size of all flows, and over 90% of the flows are smaller than that. Given the workload



(a) 99th Percentile (b) 99th Percentile

Figure 5.4: The 99th percentile FCT for small flows with varying burst size and flow rates, respectively.

distribution, this is a good threshold to use to distinguish between small and large flows. Meanwhile, increasing maximum flow rate improves the FCT modestly because more flows are tagged as high priority under a larger maximum flow rate. However, in practice, a large maximum flow rate may be abused by cloud guests to block the head-of-line of switch priority queues. A similar trend is observed for the 99.9th percentile, but to a lesser extent. Meanwhile, the impact of these parameters on the FCT of small flows at lower percentiles is even smaller, and the average FCT of large flows does not exhibit any clear pattern. Thus, both results are omitted here.

5.4 Limitations

Gaming the SRTF policies Our new policy for reducing host network queueing delay (§ 5.2.2) does not open new windows for gaming since it only requires software segmentation to be conducted earlier in the stack. However, it is possible to game our new policies for the VM scheduler (§ 5.2.1) and data center switches (§ 5.2.3), although doing so would either be very difficult or only have limited impact.

There are two potential ways to exploit our new VM scheduling policy (§ 5.2.1). First, a greedy VM may want to monopolize the shared physical CPUs by accumulating credits as it does with Xen's default policy. However, our new policy is at least as fair as the default

policy, and it makes such exploitation more difficult because the new policy allows BOOSTed VMs to preempt each other: Every VM has the same chance of getting the physical CPU, but other BOOSTed VMs can easily take the CPU back. The second issue is that malicious VMs may force frequent VM preemption to hurt overall system performance. Fortunately, by default, Xen's `rate limit` ensures that preemption cannot happen more often than once per millisecond. Moreover, § 5.3.2 shows the extra preemption overhead introduced by our new policy is rather limited because Xen's BOOST mechanism already allows fairly frequent VM preemption and our change only affects the tail cases.

For the two-class priority queueing policy (§ 5.2.3), greedy guests may break a large flow into smaller ones to gain a higher priority. However, because we *define a flow as a collection of any packets from a source VM to a destination VM*, creating N high priority flows would require N different source-destination VM pairs. Using parallel TCP connections to increase resource sharing on bottleneck links is a known problem [33] regardless of applying SRTF or not. Our new policy would only be exploited when multiple pairs of VMs are colluding together, which is not necessarily unfair because the customers have to pay for more guest VMs in this case. In fact, the fairness consideration in this scenario is a research problem by itself [70, 65, 10], which is out of the scope of this work.

10Gbps and 40Gbps networks While we impose no requirement on the core cloud network, our solution assumes 1Gbps access links. We segment large packets earlier in software to reduce host queueing delay. However, using software segmentation is not a limitation introduced by our solution; it is suggested to turn off hardware segmentation to use BQL and CoDel effectively [86]. Host queueing delay may become less of an issue with 10Gbps and 40Gbps access links because transmitting a 64KB packet only takes about 52.4μs at 10Gbps and 13.1μs at 40Gbps, in which case software segmentation may no longer be necessary.

Hardware-based solutions Hardware-based solutions may be needed to cope with higher bandwidth. As discussed, Intel's VMDq [41] may reduce host network queueing delay without any modification to the host kernel. In addition, monitoring and tagging of flows

and transmission queue scheduling can also be implemented in the NIC hardware. Because these algorithms rely on standard constructs (e.g., token bucket), and some of them are specifically designed to be hardware friendly [57], the added complexity may not be prohibitive.

Real-world workloads in public clouds To systematically determine the optimal settings for flow tagging, a detailed flow level measurement in the target public cloud is needed. It would be ideal to collect a set of flow level traces from real-world public clouds and make it available to the public so that research results may become more comparable and easier to reproduce. In lieu of such authentic workloads or traces, our synthetic workloads must suffice.

5.5 Summary

In this chapter, we mitigate the impact of virtual machine performance interference by holistically attacking three network latency problems, in the context of public clouds, with a host-centric solution that only requires modification to the infrastructure below the virtualization semantic gap. Because most existing solutions designed for dedicated data centers are rendered impractical by virtualization and multi-tenancy, we extend the classic Shortest Remaining Time First scheduling policy from the virtualization layer, through the host network stack, to the network switches without trusting guest cooperation. With testbed experiments and simulation, we show that our solution can reduce median latency of small flows by 40%, with improvements in the tail of almost 90%, while reducing throughput of large flows by less than 3%.

CHAPTER 6

Characterization of Cache-Based Cross-VM Attacks

This chapter studies the security interference between virtual machines (VMs). Essentially, the same resource contention that causes performance interference can be abused by malicious users to launch attacks. In particular, the existence of the interference can result in performance degradation attacks [76] and cross-VM information leakage [60, 66, 85, 97], which includes covert channel and side channel attacks.

In this chapter, we characterize the attacks that are enabled by abusing the shared CPU cache. Cache sharing is common in public clouds because processor sharing improves resource utilization [78]. To begin with, we study a performance degradation attack against the VMs running CPU-bound workloads. Their performance is susceptible to frequent cache misses. Using testbed experiments, we reveal the relationship between VMs' workload patterns and the effectiveness of the attack.

In addition, we explore a cross-VM information leakage attack using cache-based covert channels. This attack is also a result of abusing the contention of the shared CPU cache. The original authors of the attack have demonstrated a proof-of-concept with a channel bandwidth of merely 0.2 bits per second (bps) using shared L2 cache between co-located small EC2 instances [66]. Thus, we set out to assess its ability to do harm with more thorough experiments. Through progressively refining models of this attack from the derived maximums, to implementable channels on the testbed, and finally in Amazon EC2 itself, we show how a variety of factors impact the ability to create effective channels, and that the resulting channel may only be practical to leak small secrets like private keys.

6.1 Cross-VM Performance Degradation Attack

In this section, we discuss a performance degradation attack against virtual machines co-located on the same physical machine. This attack is enabled by the contention of shared CPU cache. Our study starts with the background of the attack and then discusses the relationship between VMs' workload patterns and the effectiveness of the attack.

6.1.1 Background

While virtual machine schedulers can guarantee the fair sharing of CPU *cycles* among competing guests, not all the hardware resources in a physical machine can be managed in the same way [72]. As a result, performance interference inevitably exits between neighboring VMs, and such interference can even be abused to launch cross-VM attacks [76, 67].

To understand such attacks, we introduce a simple scenario that has two guest VMs sharing a physical CPU. Each VM is allocated with 50% CPU time. In Xen's default work-conserving mode, schedulers allow one VM to use more CPU time than its fair share when the other VM idles. However, in public clouds like EC2, scheduling is done in non-work-conserving mode [97], which means no VM can ever uses more CPU cycles than its fair share even if no one else is using the physical CPU. Public cloud providers adopt this scheduling mode in order to make VM performance more predictable [56].

Ideally, for the two VMs in our example with non-work-conserving scheduling, if one VM runs a CPU-bound workload, its performance should not be affected by the workload running in the other VM. However, the performance of a CPU-bound task is not only determined by the number of CPU cycles allocated to it, but also by other resources such as various levels of CPU cache that are used to hide the latency in retrieving contents from the main memory. Unfortunately, for such resources, it is not easy to achieve fine-grained time-sharing as for CPU cycles. Therefore, in a public cloud, even when VMs are scheduled in non-work-conserving mode, the contention of shared CPU cache between co-located VMs on a physical machine can cause performance interference due to cache misses.

Varadarajan *et al.* demonstrate that such interference can be abused to mount performance degradation attacks [76]. Their demonstration co-locates a VM running a web work-

load with another VM running a synthetic CPU-bound workload called `LLCProbe`. What `LLCProbe` does is to scan a buffer of the same size as the shared CPU's last level cache—L2 cache in this case—in order to make itself sensitive to cache misses. This workload is similar to what we use to evaluate the impact of our new VM scheduling policy in § 5.3.2. By running these two workloads on a shared CPU, the results of their experiment show that as the request rate of the web workload increases, the runtime of `LLCProbe` surges. This is because in order for the web server to process incoming requests in time, Xen's VM scheduler would preempt the VM running `LLCProbe` to allow its neighboring VM to handle network interrupts; such operation not only incurs the overhead of context switching, but more importantly it results in a higher cache miss rate, which directly impacts the performance of `LLCProbe`. For example, at 3,000 requests/second, the web server can cause over 300% increase in the runtime of `LLCProbe` compared to the baseline that co-locates with an idle web server [76].

Now that we understand it is possible to mount a performance degradation attack against neighboring VMs by abusing the contention of the shared CPU cache, in the next subsection, we study the relationship between VMs' workload patterns and the effectiveness of such an attack using controlled experiments on a testbed.

6.1.2 Workload Patterns and Attack Effectiveness

In the demonstration of the performance degradation attack, Varadarajan *et al.* also offer a countermeasure. That is, by sending computation-intensive requests to the web server, the corresponding VM has to use more CPU time and therefore lose its privilege to preempt its neighboring VMs as frequently as before, which means the VM running `LLCProbe` will experience fewer cache misses [76].

This countermeasure indicates that the keys to the effectiveness of the attack are 1) how frequently the victim VM is preempted, and 2) the way in which the attacker VM access the shared cache. The goal of our study is therefore to *empirically* demonstrate in what conditions this performance degradation attack is most effective. We do so by varying the workload patterns of the attacker VM.

Specifically, the victim in our experiments is the VM that runs a workload similar to `LLCProbe` [76] that repeatedly scans a memory buffer of the same size as the shared L2 cache. The attacker VM runs another synthetic workload that brings frequent preemption and cache misses to the victim on the same physical machine. The effectiveness of this attack is measured by the increase in victim's runtime relative to the baseline for which there is no interference. This setup allows us to study the *empirical* sensitivity of victim's performance to cache mises.

Because what matters to the attacker is VM preemption and cache eviction, we do not need the web workload to fulfill the goal. Instead, we use another memory scanning workload that has `sleep` system calls inserted between scanning operations to control both the VM preemption rate (waking up from `sleep` may preempt the victim) and the cache miss rate (the intensity of memory scanning between `sleep` affects cache eviction).

Similar to [76], we conduct our empirical study in two scenarios:

1. Same core: the attacker VM and the victim VM are pinned on the same CPU core.

2. Same package: the attacker VM and the victim VM reside in different cores of the same CPU package with a shared L2 cache.

The difference between these two scenarios is that the co-located VMs can run in parallel when residing on different cores, while on the same CPU core they have to take turns.

Figure 6.1 shows the relative runtime increase of the victim workload, which can be considered as the effectiveness of the performance degradation attack. The baseline is generated using the same workload without the attacker VM. For both cases, the victim's runtime first increases as the sleep time between memory scans goes up and peaks around 3ms to 4ms before decreasing. This behavior demonstrates the impact of VM preemption on the effectiveness of the attack. With short sleep time, the attacker VM does not have enough credits accumulated to preempt the victim VM and evict the contents in the shared L2 cache. On the other hand, if sleeping too long, the attacker VM does not spend enough time conducting memory scanning operations, and therefore the attack becomes less effective. Only when the sleep time is around 3ms to 4ms, the attack becomes most effectively as it reaches a balance between credits accumulating and attacking.

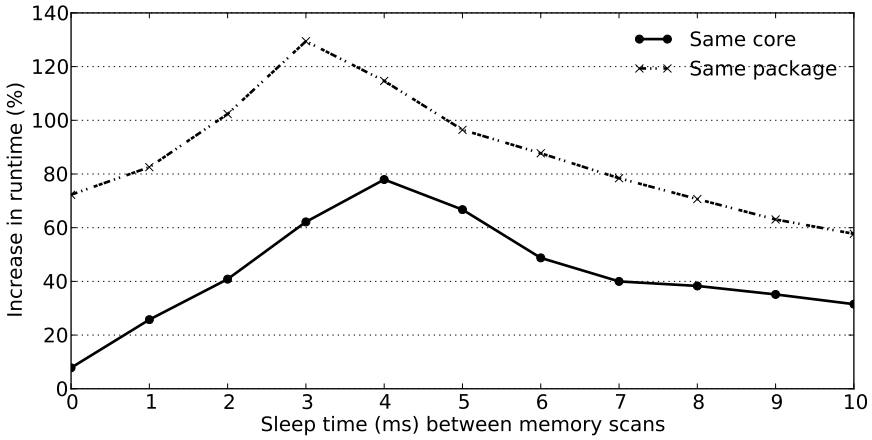


Figure 6.1: The relative increase in the runtime of a memory scanning workload attacked by a neighboring VM that also scans a memory buffer repeatedly and sleeps between scanning operations. The baseline is generated without the attacker workload. In both scenarios depicted in the figure, the victim VM and the attacker VM share at least the L2 cache.

Strictly speaking, in the scenario of "same package", the attacker VM never really preempts the victim as they run on different CPU cores. However, the sleep time does allow the attacker VM accumulates credits in order to use the physical CPU when it is most needed. Thus, the same argument about the importance of VM preemption also applies. Note that the exact value that makes the attack most effective depends on the implementation of the attack, and it also correlates with the size of the shared L2 cache. The intuition is that the attack becomes most effective when the victim and the attack conduct their memory scanning operations in lockstep.

Additionally, the effectiveness of the attack in these two scenarios also differs significantly. This is because when the victim VM and the attacker VM are running on the same CPU core, the attacker VM can only evict the cache contents when the victim is first preempted. After a full eviction, the attacker's subsequent operations do not affect victim's performance until the victim VM regains the control of the shared CPU core. In other words, many of attacker's memory scanning operations fail to translate into real impact on the victim. We also repeated the same experiment with the two VMs running on different CPU packages that share nothing but the memory bus. In comparison, the runtime of the victim workload is only increased by up to 13.8% in this third scenario. It confirms the intuition that this performance degradation attack requires shared CPU cache to be effective.

In summary, while cache miss is the main reason why the victim's workload slows down, the frequency of VM preemption controls the effectiveness of the attack. Importantly, the negative impact of the attack becomes the worst when the memory access patterns of the victim's and attacker's workloads synchronize. In Chapter 7, we rely on this observation to construct a mitigation mechanism.

6.2 Cross-VM Covert Channel Attack

In this section, we explore another cache-based cross-VM attack called covert channel attack. This attack is an instance of information leakage attacks that abuse the shared L2 cache. Side channel attacks also belong to this category, but we consider it as a *special case* of the covert channel attack because both attacks share the same principle.

Basically, both of these two types of information channels have a destination VM learning information secretively from a source VM. The difference between the two is that for covert channels, the attacker has a full control over the information encoding at the source VM. However, the attacker of a side channel has no such control, and it is the leaking software at the source VM that determines its (unintentional) information encoding scheme. In other words, a side channel is "in fact a covert channel without conspiracy or consent" [74]. In comparison, the covert channel model allows us to quantify the impact of information leakage in controlled experiments, and we use this more general model for the rest of this chapter.

6.2.1 Background and Related Work

To fully understand the performance and limitations of the L2 cache covert channel, we first present some background information on cache-based covert channels in general and their ability to do harm.

6.2.1.1 Cache-Based Covert Channels

Ristenpart *et al.* first introduced the concept of cross-VM covert channels [66]. Their basic idea is to construct certain patterns of contention on the hardware resources shared by two co-located VMs and use the contention patterns to encode information. For example, to send a single bit via a shared hard disk, attackers may let both the sender and the receiver VMs operate on large files concurrently for a specific period of time. During that time, the sender can encode information by reading files (bit one) or doing nothing (bit zero). At the same time, the receiver can distinguish these two states by timing its own disk operations and decode the information.

Based on this idea, Ristenpart *et al.* implemented three proof-of-concept covert channels on EC2: a 0.006bps channel using memory bus contention, and 0.0005bps channel using hard disk contention, and a 0.2bps channel using L2 cache contention [66]. As a follow up, Okamura *et al.* designed and evaluated a new attack that uses the load of a shared CPU to encode information [60]. In addition, Zhang *et al.* monitor the patterns of L2 cache usage within a guest domain to build a classifier of the usage patterns to check if there are other VMs sharing the same physical machine [96].

While the number of possible channels is open-ended, we have opted to focus on *L2 cache covert channels* as described in the prior work [66] for a variety of reasons: (i) it arguably has the highest potential bit rate as the time needed to make a contention measurement with modern L2 cache is on the scale of milliseconds, but the reported rate in prior work was very small, (ii) prior work only discussed such channels in brief and did not provide an in-depth analysis of the mechanisms involved and their effects, both qualitative and quantitative, in various settings, (iii) reproducing these results bolsters confidence in previous experimental results and allows for an exploration of changes to EC2 allocation and placement strategies since the initial work.

In particular, the L2 cache-based information encoding scheme proposed by Ristenpart *et al.* can be summarized as follows [66]. All the cache lines are divided into two subsets (a and b). To send a bit, the sender evicts the receiver's cache contents from the cache lines corresponding to one subset and leave the other untouched by accessing the memory

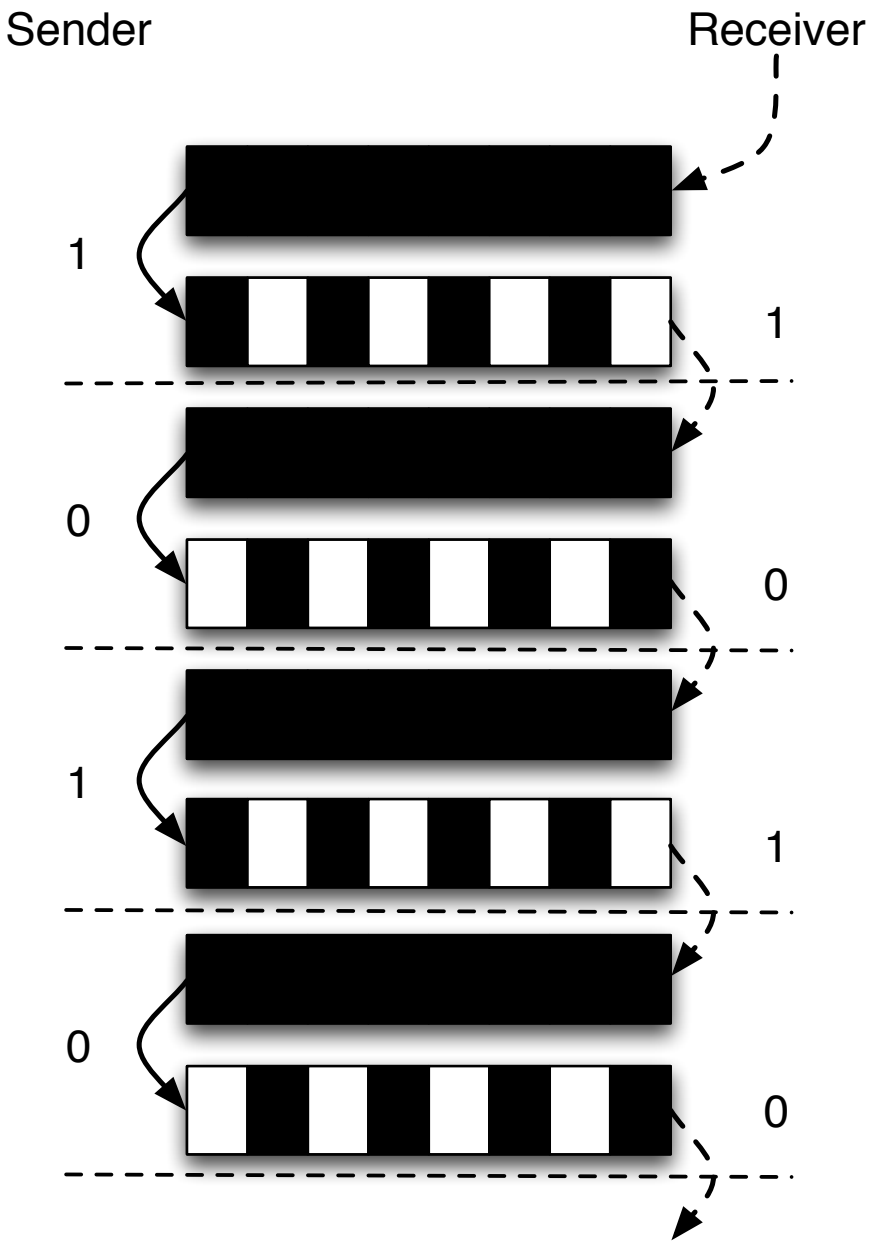


Figure 6.2: An illustration of a covert channel using L2 cache to encode information. For each bit, the sender evicts half of the cache lines from the L2 cache saturated previously by the receiver (solid lines). The receiver then decodes the information by measuring the timing difference in accessing different subsets of the cache (dashed lines).

addresses mapped to the chosen cache lines. Then, the receiver can decode the information by comparing the timing in accessing the two subsets separately, and if subset *a* takes significantly longer to read than subset *b*, it is bit one; otherwise it is bit zero. This process is illustrated by Figure 6.2.

6.2.1.2 Assessment of the Bit Rate Required to do Harm

Obviously, a covert channel with the bit rate of 0.2bps cannot be leveraged to exfiltrate large data records. Take the numbers from the Digital Forensics Association for example. From 2005 to 2009, the number of lost records in data breaches ranged from 1 to $130,000,000$ with an average of $387,926$ [24]. While the report did not indicate the size of a lost record, we can explore the variety of record types to understand the practical impact of the channel bit rate. For example, assume that each record contains the information from the first track of a credit card magnetic strip [81] (i.e., name, number, dates, etc.). If we use a 32-bit encoding scheme (while more effective encoding schemes may exist, we simply choose the easy one to implement UTF-32), the size of a single record is $2,528$ bits. It would take 3.51 hours to leak this piece of information with a 0.2bps covert channel. Transferring all the information contained in a average data breach would take more than a century at 0.2bps bit rate. If the records leaked are medical data, each single record may contain $1,024,000$ bits [14]. In this case, even leaking a single record would take the 0.2bps channel as long as 1422.22 hours.

Therefore, covert channels with this scale of bit rate may only be useful in leaking small cryptographic secrets. For example, the 2048-bit private key owned by the author contains $1,743$ bytes. It will take about 20 hours to leak this key with the 0.2bps L2 cache covert channel. However, if using a channel with the maximum bit rate of 262.47bps, as described in § 6.2.2, leaking the same private would only take about 53 seconds. It should be noted that these are simple illustrations and one does not always need to leak an entire secret key to compromise it [20]. Often, even a few bits may suffice to reconstruct a cryptographic secret, and in such cases low bandwidth covert channels also become important.

Given the 1000x difference between the bit rate achieved in existing work on EC2 [66] and the maximum bit rate we calculated, we find ourselves compelled to ask:

"What is the maximum achievable bit rate in practice over cross-VM covert channels? What factors influence the bit rate achievable in cross-VM covert channels?"

6.2.2 Naive Quantification of Channel Bit Rates

Before discussing the design and implementation of the L2 cache covert channels, we present a back-of-the-envelope calculation for its maximum bit rate. The numbers used for the discussion in this section are derived with the following L2 cache specifications:

- L2 cache size: 6MB
- L2 cache line size: 64B
- L2 cache associativity: 24-way

In addition, we use *7 nanoseconds* as the time to fetch from the L2 cache, and *100 nanoseconds* as the time to fetch from main memory [21]. With these numbers, estimating the time to transmit a symbol on the covert channel is straightforward. The number of bits that can be encoded into a symbol is an open question and depending on the CPU design, it may range from one to $log(\#\text{ of cache lines})$. In order to compare with existing work [66], we assume that each symbol contains just one bit of information. Thus, we can use the same information encoding scheme described in the background section.

In the ideal case, the minimum time to send a bit using L2 cache is the sender's write time (T_w) plus the receiver's read time (T_r). Thus, the maximum bit rate would be $1/(T_w + T_r)$. And this ideal case can be summarized using Protocol 1($P1$). While this initial protocol is very simple, it evolves as the channel environment becomes more and more complicated.

Protocol 1 ($P1$)

1: The sender and receiver both allocate a buffer with the same size as the L2 cache. Then the receiver accesses its own buffer to fulfill the cache lines (one time initialization).
2: Repeat Step 3 and 4 sequentially for each bit.
3: The sender accesses subset a (or b) to send bit one (or zero) (T_w).
4: The receiver accesses subset a and b separately to decode the information (T_r).

Now we just need to estimate T_w and T_r to calculate the maximum bit rate. However, considering the fact that L2 cache is usually shared by both code and data, these two numbers become difficult to compute. In addition, modern CPU advanced features like prefetching that further complicate this task. We ignore these complications for the moment, and assume T_w and T_r can be naively estimated as follows:

$$\begin{aligned} T_w &= 6MB/64B/2 * 100ns \approx 5ms \\ T_r &= 6MB/64B/2 * (100ns + 7ns) \\ &\approx 5ms + 0.34ms = 5.34ms \end{aligned}$$

Here the cache size is divided by the cache line size because, in order to fill up a cache line, it only needs to be accessed once by any byte within it. As a result, the maximum bit rate using the L2 cache in this ideal case would be

$$1bit/(5ms + 5.34ms) \approx 96.71bps$$

If more realistic numbers are desired, the complications discussed above must be put back into the calculation. To solve this problem, instead of looking into the hardware design manual to refine the estimation, we simply implement the basic operations of Step 3 & 4 in $P1$ on a testbed machine, which has the same L2 cache configuration as specified earlier in this subsection and a 2.83GHz clock speed, and then profile T_w and T_r by executing these operations. The profiling results for T_w and T_r are 1.47ms and 2.34ms, respectively. Therefore, the maximum bit rate of L2 cache covert channel on testbed machines would be

$$1bit/(1.47ms + 2.34ms) \approx 262.47bps$$

6.2.3 Achievable Bit Rates on the Testbed

The channel bit rate estimated with protocol $P1$ is unrealistic and, in particular, the idealized model overlooks the following factors:

- The operations of the sender and the receiver cannot be synchronized perfectly.

- There is other overhead associated with the channel program, such as process creation and destruction, that reduces the channel bit rate.

Specifically, the sender and receiver, as two separate VMs, have no method to synchronize perfectly in a way that the receiver's read can follow immediately after the sender's write. To solve this problem, Ristenpart *et al.* used a busy loop at the receiver side to wait for the sender until the receiver perceives a large jump on its CPU counter [66]. Clearly, this solution implies two requirements: a) the two VMs share the same core at least for the next bit ($R1$), and b) their VCPUs are pinned to the shared core ($R2$); otherwise the receiver would not be able to notice the timing of the sender's operations. Now we derive a new Protocol 2 ($P2$) that is applicable in a laboratory environment.

Protocol 2 ($P2$)

1: One time initialization. Same as $P1$
2: Repeat Step 3 and 4 sequentially for each bit.
3: The sender accesses subset a (or b) to send bit one (or zero) (T_w). Then it goes to sleep for T_s.
4: The receiver busy loops until CPU counter jumps by at least N_j. Then it accesses subset a and b separately to decode the information (T_r).

For the second problem, the overhead can only be estimated by actually running the covert channel on the testbed. To implement this new protocol and estimate its potential bit rate, we just need to determine the value of T_s and N_j. The minimum value required for T_s should be large enough to allow the receiver to finish its operation, i.e., $T_s \geq T_r = 2.34ms$. Consider the variability of T_r, we conservatively choose $T_s = 3ms$ as the minimum sleep time. In addition, N_j should be large enough to cover the cycles needed to for the sender to finish its cache operation. Given a CPU with a 2.83GHz clock speed, $N_j \geq 1.47 * 2830000 = 4160100$. Consequently, a practical bit rate that is possible to be realized in a laboratory environment that satisfies requirements $R1$ and $R2$ would be:

$$1bit/(1.47ms + 3ms) \approx 223.71bps$$

The difference between 223.71bps and the theoretic maximum 262.47bps can be considered as the *synchronization overhead* in practice. We implement this channel using pro-

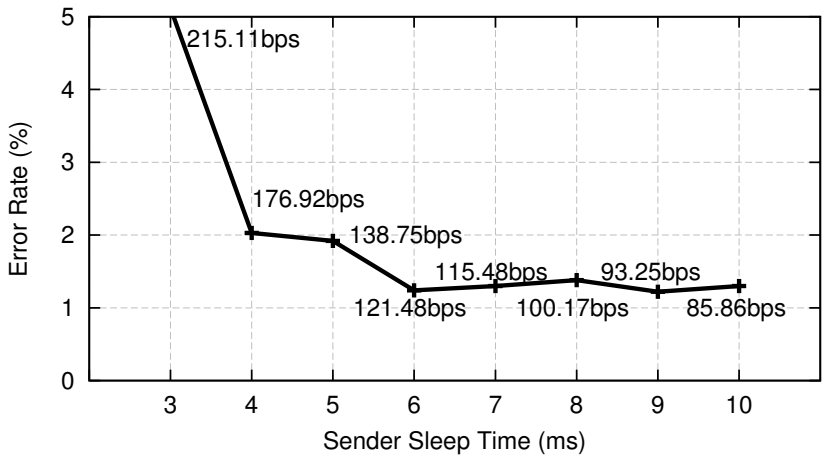


Figure 6.3: The error rate for the covert channel built on the testbed drops as the sleep time increases. When the sleep time is larger than or equal to 6ms, the error rate becomes stable.

tocol *P*2 on our testbed by transmitting a 64-byte text 100 times. Using the preceding parameters, a bit rate of 215.11bps is obtained with an 5.12% error rate on average. The loss in bit rate is due to the *overhead of non-cache-related code*, such as process creation/destroy, and the code that coverts the source of information into raw bits for the sender.

In order to reduce the error rate for the channel, we can increase the value of T_s because the uncertainties generated by the process/VM scheduling algorithms and other environmental factors may cause the channel out-of-sync occasionally. Figure 6.3 depicts the relationship between sender's sleep time and the channel error rate. The error rate drops as T_s increases and stabilizes when the sleep time is larger than or equal to 6ms. As expected, when the sleep time T_s of the sender approaches its minimal, out-of-sync errors are more likely to happen. On the other hand, the impact of sleep time to the channel error rate diminishes when T_s becomes large enough to tolerate the variability of T_r.

Before we close the discussion about channel bit rate in the laboratory environment, there is still one difference to be noted between *P*2 and the protocol proposed by Ristenpart *et al.*: the receiver also sleeps briefly every iteration in their version [66]. According to Ristenpart *et al.*, the rationale behind the sleep is for the receiver "to build up credit with Xen's Scheduler" [66] because its busy loop may eat up scheduling credits quickly so that the receiver is less likely to be scheduled on time when there is a third VM (or more) sharing the same core. To give a complete picture, we define this version as Protocol 3 (*P*3).

Protocol 3 ($P3$)

1: One time initialization. Same as $P1$
2: Repeat Step 3 and 4 sequentially for each bit.
3: The sender accesses subset a (or b) to send bit one (or zero) (T_w). Then it goes to sleep for T_{ws}.
4: The receiver sleeps for T_{rs}, after which it busy loops until its CPU counter jumps by at least N_j. Then it accesses subset a and b separately to decode the information (T_r).

In our previous effort in building this covert channel, we started with this version of the protocol. And in the same laboratory environment, we only obtained a bit rate of around 30bps when $T_{ws} = 10ms(\text{or } 20ms), T_{rs} = 3ms$ (both choices of T_{ws} produce a similar maximum bit rate), while the expected bit rate should be close to

$$1bit/(1.47ms + 10ms) \approx 87.18bps$$
$$1bit/(1.47ms + 20ms) \approx 46.58bps$$

based on the above analysis. To explain this huge difference, we first look at the time to transfer one bit for 30bps case: $1s/30bps \approx 33.3ms$. Not surprisingly, this value is close to the maximum time slice that Xen's credit scheduler allows a virtual CPU to occupy a physical CPU [88]. Thus, the explanation for the anomaly is that when both the sender and receiver are put into sleep for each bit, the receiver always wakes up first to start its busy loop, during which the sender comes back to be runnable. If at this moment the sender preempts the receiver to run immediately, everything stays as expected. However, in reality, the receiver may have accumulated enough credits to be awarded an equal priority (*over*) as the sender. Therefore, the sender may not be able to preempt the receiver but blocked for another 30ms in the worst case. While this scenario does not happen every iteration, it indeed happens frequently enough to reduce channel bit rate to be around 30bps despite the value of T_{ws} being 10ms or 20ms. And this phenomenon persists with protocol $P3$ until $T_{ws} > 30ms$, which effectively reduces the expected bit rate to be $\leq 30bps$. Consequently, in practice we suggest not to use protocol $P3$ unless a workload can justify its usage with an improved error rate.

6.2.4 Achievable Bit Rates on EC2

According to the preceding analysis and experiment results, the difference in bit rate between the derived maximum and the one reported in prior work [66] is more than three orders of magnitude. Given that we have not intentionally optimized our code either, we speculate that the following factors may contribute to this huge gap:

- Hardware specification
- Workloads in other VMs on the same physical machine
- Hypervisor configuration, i.e., scheduling policies
- Design of the protocol

In this subsection, we analyze these factors in order and verify our speculation by implementing the L2 cache covert channel on EC2.

Hardware Specification The small EC2 instances we use have the same L2 cache specification as our lab machines. However, the CPUs on our lab machines have a higher clock speed of 2.83GHz than that of 2.66GHz on the EC2 instances we use. Even if considering that the CPUs on EC2 may be one or two generations older, the impact of hardware specification on channel bit rate still should not be more than 10%. Moreover, one may notice that EC2 uses 40% CPU cap on small instances, which effectively reduces the clock speed by 60%. We discuss this problem in more detail when it comes to the hypervisor configuration.

Workloads Given the design of the protocols $P1$ to $P3$ that the sender does not verify if the receiver has received the current bit and whether the received bit is correct, workloads on other VMs sharing the same hardware would only increase the error rate of the channel. Meanwhile, if the VMs with the workloads also share the same core as the covert channel VMs, then they will reduce the channel bit rate because they steal CPU cycles from the covert channel.

To support this argument, we introduce a third VM on our testbed to share the same core with the other two VMs used for evaluating $P2$. On the third VM, we deploy a web applica-

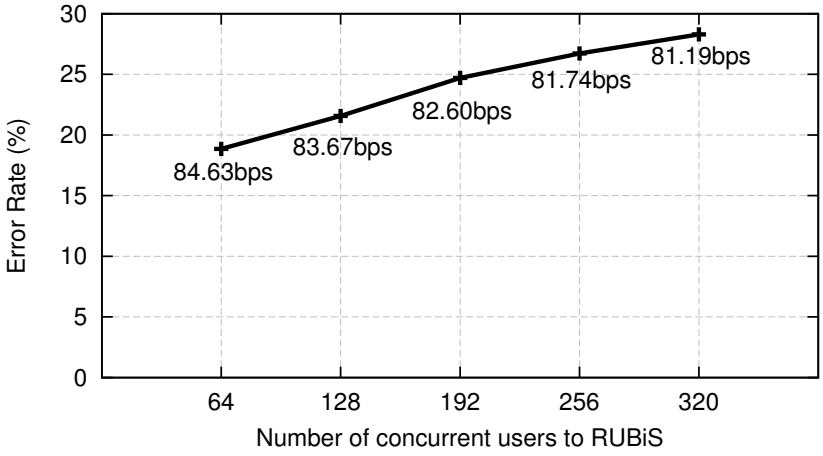


Figure 6.4: In the laboratory environments, when running a web application on a third VM sharing the same core as the covert channel, the channel error rate increases, and bit rate decreases slowly as the number of users to that application grows.

tion called RUBiS, which is an eBay-like online auction application that is used extensively as a benchmark in the research community [16, 63]. The default workload of RUBiS has a parameter called "number of clients per node" that controls the number of concurrent users accessing the application. We vary this parameter and run a covert channel in parallel on the other two VMs using $P2$ with 10ms sleep time. The resulting error rate and bit rate are plotted in Figure 6.4. As the figure suggests, the error rate increases slowly as the number of concurrent users grows. At the same time, because more concurrent users means more cycles needed to process their requests, the channel bit rates also slowly decreases. However, compared to the original bit rate 85.86bps without third-party workload, the biggest drop in bit rate is no more than 6%. Thus, we speculate that the impact of workloads themselves on the covert channels on EC2 is very limited.

Hypervisor Configuration Suppose the Xen hypervisor on EC2 has the same credit scheduler with identical parameters as the standard Xen scheduler. When protocol $P3$ is applied, the bit rate can be reduced by an order of magnitude. In addition, if we examine the CPU usage in small instances on EC2, a 40% cap should be noticed. This cap may effectively reduce the channel bit rate by at least 60% over an extended period of time. Up to this point, the raw bit rate should have been reduced to about 10bps. Furthermore, the

VCPUs used by EC2's small instances are not pinned to any specific physical cores. Using the CPUID instruction, which does not seem to be virtualized on EC2, small instances suggest four cores on a single physical machine, and VCPUs constantly migrate to different physical cores on the scale of milliseconds to thousands of milliseconds. It means at any moment, two VMs co-located on the same physical machine may not be able to communicate via the L2 cache simply because they are not running on the right cores. This policy together with the 40% cap violate both $R1$ and $R2$, which are required to ensure the high bit rate for the testbed experiments. They would also significantly affect the design of the protocol to be used on EC2.

Protocol Design Ristenpart *et al.* use multiple samples for a single bit to compensate the negative impact of core migration [66]. Thus, the resulting bit rate depends on the number of samples taken per bit, given protocol $P3$. To continue the above rough estimation, it is reasonable to obtain a bit rate of 0.2bps should we use 50 samples per bit.

To sum up, the major factors that may significantly affect the channel bit rate on EC2 include Xen's scheduling algorithm, the 40% CPU cap, and non-pinned VCPUs (core migration), all of which are environmental factors, and the design of the communication protocol, which is an artifact of the environmental factors. In the follow subsection, we describe our experiments as concrete examples to explain how does the environmental factors reduce the channel bit rate by at least two orders of magnitude in practice on EC2 with our refined protocol.

6.2.4.1 EC2 Co-location Revisited

Ristenpart *et al.*'s work explored the relationship between EC2's instance placement and its network configuration and designed a method to co-locate the attacker's VM to its victim with a probability better than a brute-force approach [66]. In our case, we care about the maximum achievable channel bit rate on EC2, so only two VMs both controlled by an attack need to be co-located, And it is a much easier task than co-locating with a targeted victim VM. In this subsection, we start with the experience learned from the existing work [66], then share our revised experience for achieving and verifying VM co-

location in the case that Amazon may have changed its polices and algorithms in response to that paper.

To begin with, because it is believed that VMs belong to the same account will never be placed on the same physical machine on EC2 [66], we start with two accounts, each of which allows 20 concurrently running VMs by default. Because of the "parallel placement locality" [66], which says VMs launched simultaneously at the same availability zone (which roughly corresponds to a data center) are more likely to be co-located, we launch 20 small instances for each account roughly at the same time. Then, the first hop IP (internal to EC2) of each VM's outbound route is checked. If any pair from the two accounts share the same first hop, they are believed to be co-located on the same physical machine [66].

Based on our experiments conducted in April 2011 on EC2's east region, two observations worth a discussion. First of all, for different accounts, the availability zones with the same name (e.g., us-east-1b) may not refer to the same physical infrastructure. In other words, the mapping between the name of availability zones and their physical locations is different from account to account. Fortunately, the actual mapping seems to be fixed once an account is created, and the mapping can be extracted by combining the information provided by several standard EC2 commands [34]. Because we are not the first to discover this phenomenon or the matching technique, and this technique is also out of the scope of this dissertation, we encourage curious readers to read the referred blog post [34]. In addition, while for each trial we roughly get 5 VM pairs with the same first hop out of the 20 by 20 candidates, not all of them can be confirmed to be co-located by running the L2 covert channel to be described later in this section. Meanwhile, we find that the pairs of VMs that allow covert channel communication have a much lower round trip time (RTT) if we do a tcptraceroute against the VMs within the same pair than that for the pairs failing our covert channel attack (0.06ms vs. 0.2ms).

Interestingly, among all the confirmed co-located VM pairs, we find one pair that belong to the same account. This pair was created when EC2 was experiencing a major outage on its east region [26], and it never happened again after the outage was fixed. Therefore, we speculate that during the outage when not all the physical machines were usable, EC2 tried to fulfill as many custom requests at the cost of a reduced guarantee of fault tolerance.

6.2.4.2 L2 Cache-Based Covert Channel On EC2

Once the VM co-location is verified, we run our own version of the L2 cache-based covert channel on EC2 to explore its potential bit rate. As discussed above, none of the existing protocols ($P1$ to $P3$) work well on EC2 due to the environmental factors related to the hypervisor configuration. As a response to these environmental factors, we present a refined protocol to explore and explain the empirical bound on the bit rate of the L2 cache-based covert channel on EC2.

Recall that for protocol $P1$ to $P3$, a busy loop at the receiver's side is used for synchronization. However, given the 40% CPU cap alone, this method is guaranteed to lose information because blind spots will appear in the receiver's lifetime when the receiver VM is preempted by the hypervisor, and these blind spots can be large enough (60% of CPU time, hence on the scale of tens of milliseconds given the 10ms ticks of Xen's credit scheduler) to allow other VMs (or the sender itself) to overwrite the cache content before the receiver decodes any useful information. To solve this problem, we eliminate the synchronization mechanism entirely and simply busy loop with the cache operations at both sides of the channel. Here we define a *valid measurement of cache contention* as one side of the channel captures the time difference in accessing the cache subsets after the other side evicts certain cache lines. Then an invalid measurement would be that one side of the channel accesses the cache before the other side touches it, i.e., no time difference would be perceived. Consequently, the design of $P1$ to $P3$ implies that all the measurements taken during the channel lifetime are valid.

This design decision is made based on the fact that while when a valid measurement of cache contention will happen is unpredictable, once it happens the verification of the measurement is very accurate because the cache interference introduced by other processes or VMs are assumed to be uniformly distributed so that the difference between the time it takes to access the evicted cache subset and the non-evicted one remains noticeable (1.47 vs. 0.87ms in practice) on the scale of milliseconds.

However, this brute-force strategy leaves an obvious question: given that there is no guarantee for a valid measurement of cache contention to happen, even if we can take

valid measurements once in a while, how do we verify whether the current bit has been transferred and when the sender and receiver should synchronously move on to the next bit. Even worse, the core migration on EC2 can make the cache contention between two given co-located VMs never happen in the worst case. Thus, the short answer to this question is that there can be *no* guarantee because we are trying to send information reliably on a unreliable channel, which is a problem similar to the TCP handshake protocol [40]. In fact, the protocols ($P1$ to $P3$) we have discussed so far are all one-way protocols, which means the receiver has no way to acknowledge the sender for the transmission, but it is not a problem when both sides can synchronize each other with the busy loop. Fortunately, if a valid measurement of cache contention happens to the receiver, it should have evicted the sender's content from the cache. The next time the sender takes a measurement, it will also have a chance to take a valid measurement. That means we can use a similar method to time the cache operations at the sender's side as a pseudo-acknowledgement of the transmission. However, it suffers from the same problem that it has no guarantee to happen. To break this deadlock, similar to the scheme used by Ristenpart *et al.* [66], we simply repeat this procedure multiple times for each individual bit for transmission to increase the possibility of success. This new algorithm is summarized as Protocol 4 ($P4$).

Protocol 4 ($P4$)

1: One time initialization. Same as $P1$
2: Repeat Step 3-4 and 5-6 concurrently for each bit.
3: The sender accesses subset a (or b) to send bit one (or zero) (T_w). Repeat this process until obtains N_w valid measurements. After each measurement, despite valid or not, the sender sleeps for T_{ws}.
4: The sender accesses the cache subset opposite to the one used in Step 3. Repeat this process until an invalid measurement happens.
5: The receiver accesses subset a and b separately (T_r) to decode the information. Repeat this process until obtained N_r valid measurement. After each measurement, despite valid or not, the receiver sleeps for T_{rs}.
6: The receiver accesses the same subset used in Step 5. Repeat this process until an invalid measurement happens.

Before we move to the bit rate analysis with this protocol, the rationale behind Step 4 & 6 and the use of T_{ws} & T_{rs} should be explained first. Because contention measurements are now repeated multiple times for each bit, depending on the choice of N_w and N_r, either

the sender or the receiver may get stuck in Step 3 and 5 respectively due to unexpected core migrations. As a result, Step 4 and 6 are used to check if the measurement loops used by the other side of the channel for the current bit have finished. For example, if Step 6 succeeds for the receiver, it means the sender is also likely to be working on Step 4, which implies they both can move on to the next bit. Again, there is still no guarantee that the success of Step 4 and 6 indicates that the sender and the receive have been synchronized for the next bit since core migration at that point can lead to the same perception, their presence increases the chance of being synchronized while their overhead on the bit rate is limited as the number of samples taken per bit increases. In addition, the sleep time of T_{ws} & T_{rs} are used to avoid repeating too fast to miss the opportunity of cache measurements, and 1ms will suffice. Please note that because for protocol $P4$ the sender and the receiver have a similar busy loop and also share the same sleep time, the sender's over-blocked problem caused by Xen's scheduler no long exists.

We implement $P4$ on EC2 with two co-located VMs found by the method described in the last subsection. To analyze the potential bit rate, we first profile the performance of basic cache operations to obtain T_w and T_r as follows:

$$\begin{cases} T_{w1} = 2.67ms, T_{r1} = 3.73ms & \text{if contention happens} \\ T_{w0} = 1.10ms, T_{r0} = 2.09ms & \text{otherwise} \end{cases}$$

Please note that the profiling result does not precisely follow the theory. For example, in the case of no contention happens, T_{r0} should be two times of T_{w0}. This slight miss match is just an artifact of a profiling-based method itself, which is the best we can do for a black-box system like EC2, and its impact on the analysis result should be limited. Meanwhile, the numbers may vary on a small scale if profiled with a different pair of co-located VMs.

In our implementation, the receiver is used to constantly monitor the cache content because while there is no guarantee on a successful bit transmission, once it succeeds the chance for the receiver to get a flipped bit is almost zero. Also, because the operations of the sender's and receiver's are always concurrent, we use the sender's side to analyze the channel bit rate in the following discussion.

In the ideal case, core migration never happens during the transmission so that every

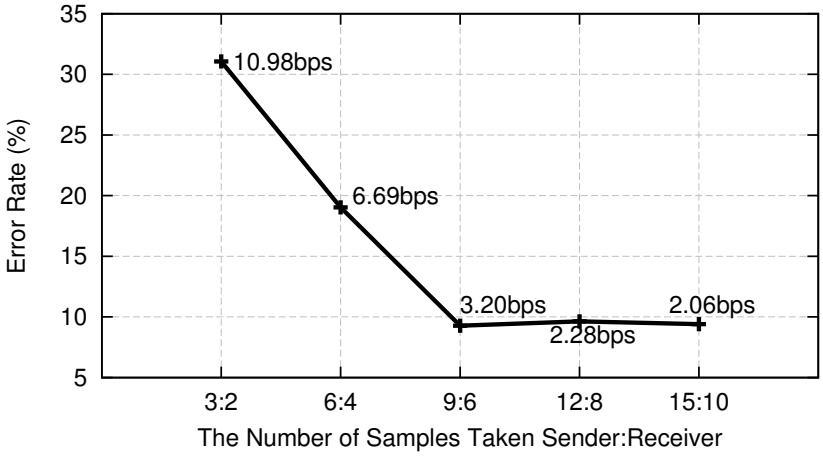


Figure 6.5: The error rate of the covert channel built on E2 using protocol $P4$ drops as more more samples ($N_w : N_r$) are taken for each bit. After $N_w : N_r = 9 : 6$ no significant benefits can be gained by increasing the number of samples.

	Mean	Median	Max	Min
Bit Rate	3.20bps	3.75bps	10.46bps	1.27bps
Error Rate	9.28%	8.59%	28.13%	0%

Table 6.1: Basic statistics of the channel bit rate and error rate on EC2.

measurement would be valid. This overly optimistic assumption results in the following estimation of the ideal channel bit rate on EC2:

$$bitrate = 0.4 * 1bit/(N_w * (T_{w1} + T_{ws}) + T_{w0})$$

Again in the best case where we choose $N_w = 1$, the maximum bit rate would be

$$0.4 * 1bit/(2.67ms + 1ms + 1.10ms) \approx 83.86bps$$

However, this estimation is clearly unrealistic. Because it implies $N_r = 1$, but $T_{w1} \neq T_{r1}$ means N_w and N_r cannot take the same value. Instead, using $N_w : N_r \approx T_{r1} : T_{w1} \approx 3 : 2$ would be appropriate.

Intuitively, smaller values of N_w tend to result in higher error rates because the sender and the receiver are more likely to be out-of-sync. Figure 6.5 demonstrates the relationship between the error rates and the number of samples ($N_w : N_r$) with corresponding bit rate.

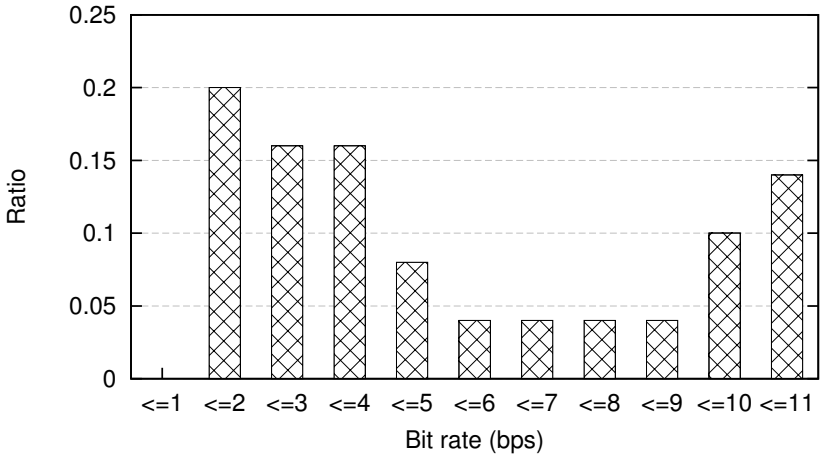


Figure 6.6: The bit rate distribution for the covert channel in EC2 using protocol P4.

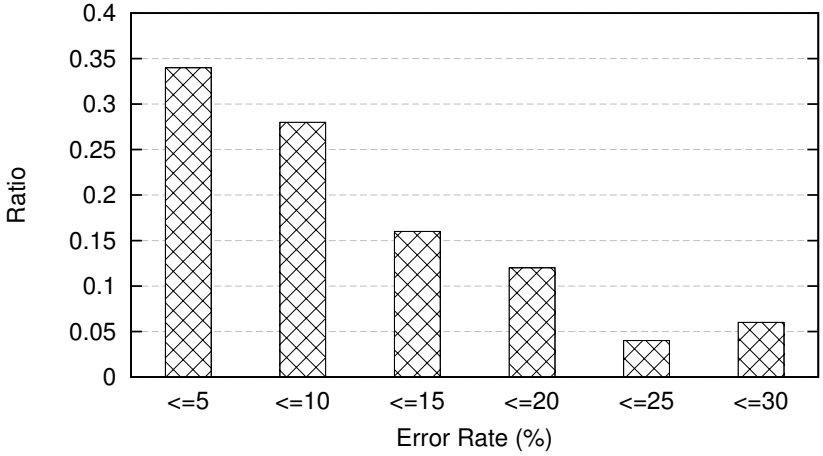


Figure 6.7: The error rate distribution for the covert channel in EC2 using protocol P4.

While smaller values yield higher error rates, once $N_w \geq 9$, the error rate becomes stable.

As a result, we choose $N_w = 9, N_r = 6$ for further experiments. Figure 6.6 and Figure 6.7 shows the distributions of the bit rate and error rate for this pair of parameters, respectively. Table 6.1 shows some basic statistics about these two metrics. The impact of the environment factors on these metrics is discussed in the following paragraphs.

First of all, given $N_w = 9$, if every cache measurement is valid, the expected channel bit rate would be

$$0.4 * 1bit/(9 * (2.67ms + 1ms) + 1.10ms) \approx 11.72bps$$

Considering other overhead associated with the covert channel program, the maximum bit rate of 10.46bps we obtain through experiment is very close to this expected number. Obviously, there two components that reduce the expected bit rate from 83.86bps to 11.72bps: the 40% CPU cap, which effectively reduce the bit rate by 60% if the channel runs for an extended period of time, and the repeated sampling, which is designed to deal with the core migration and other uncertainties in VM scheduling.

Even worse, for the same reason, not every cache measurement can be valid. Thus, we need to obtain the average percentage of time that the sender spends on valid cache measurements among all the measurements that have been made during the transmission period. This number should be environment dependent, and it can be only obtained through profiling. With the trials conducted on the same EC2 instances, we divide the cache measurements into three category using the timing of each measurement: valid measurements, invalid measurements, and the measurements made when VM gets preempted. While the first two categories are self-evident, the last one contains the measurements that take more then 3ms to make, so that they are believed to be taken when corresponding VM is preempted by the hypervisor. The percentage of each category is listed as follows:

$$\begin{cases} 10.5\% & \text{Valid cache measurements} \\ 39.7\% & \text{Invalid cache measurements} \\ 6.5\% & \text{Measurements made when VM preempted} \end{cases}$$

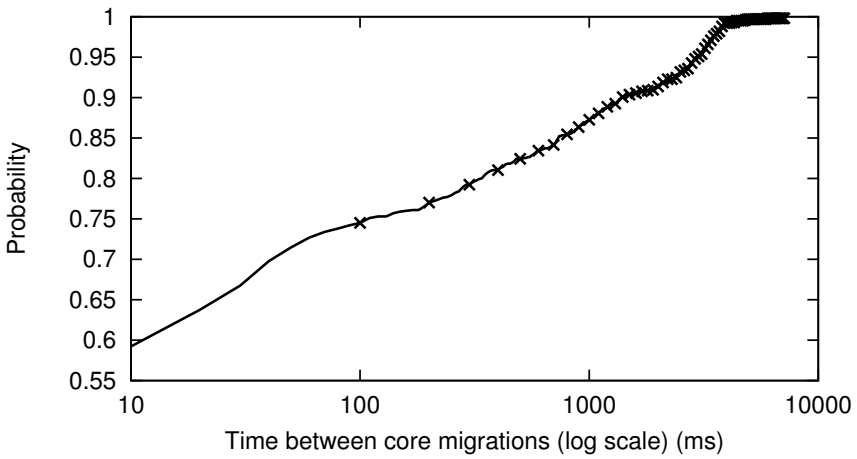


Figure 6.8: The CDF of the time between core migrations. About 50% of the time a VCPU stays on the same core for no more than 10ms, while about 25% of the time a VCPU stays on the same core for more than 100ms.

As we can see, only 10.5% of the transmission time is actually spent on valid cache measurements. It explains why the experiments shows a bit rate of 3.20bps on average. These three categories make up 56.7% of the transmission time. The rest of it is in fact the 1ms sleep time after each measurement. As discussed before, this sleep time is used to avoid polling too fast for both sides. Its removal would significantly increase the error rate. On the other hand, reducing the sleep time for each iteration would not improve the channel bit rate in a meaningful way because this large amount of sleep time is caused by the large percentage of invalid cache measurements, which in turn is caused by the uncertainties in VM scheduling such as core migration.

To understand why valid cache measurements only consist 10.5% of the transmission time, we profile the frequency of core migrations during our experiments. Note that because to some extent a core migration is triggered by the execution of instructions, the profiling process itself would impact the profiled frequency. In addition, the profiling is conducted along side the covert channel communication; its resolution is therefore constrained by the polling frequency of the target channel. Consequently, the results should not be viewed as a characterization of the scheduling properties for the underlying scheduler, but just as a concrete example to illustrate the impact of environment factors to the channel bit rate. Figure 6.8 shows the distribution of the time between core migrations. As we can see,

almost 50% of the time a VCPU stays on the same core for no more than 10ms. This frequent migration will cause a large number invalid cache measurements. Meanwhile, about 25% of the time, a VCPU stays on the same core for more than 100ms, which gives the channel a plenty of time to make valid measurements. This perception also explains why the channel bit rate in Figure 6.6 shows a bimodal distribution.

To sum up, for this particular cache-based cross-VM covert channel attack, while it can achieve a modest bit rate on a fully controlled testbed, its channel bandwidth between real EC2 instances only allows to leak small secrets like private keys. Meanwhile, similar to the performance degradation attack discussed in § 6.1, in order to make this covert channel attack relatively effective, attackers also need to rely on frequent VM preemptions to enable alternating usage of the shared cache between the attacker VM and the victim VM.

6.3 Summary

In this chapter, we study the security interference between the virtual machines that share CPU L2 cache. We characterize a performance degradation attack by revealing the relationship between VMs' workload patterns and the effectiveness of the attack. We also explore a covert channel attack by providing a quantification of its bit rates and accessing its ability to do harm. We expand the scope of the pioneering work for this threat [66] by progressively refining the models of cross-VM covert channels from the derived maximums, to implementable channels on the testbed, and finally in Amazon EC2 itself. We show how a variety of factors impact the ability to create effective channels. While a covert channel with a considerably larger bit rate than previously reported is demonstrated, we assess that even at such an improved rate, the harm of data exfiltration from these channels is still limited to the sharing of small secrets such as private keys. Importantly, the effectiveness of these cache-based cross-VM attacks rely on frequent VM preemption, and this observation is the foundation of our mitigation technique discussed in Chapter 7.

CHAPTER 7

A Host-Centric Approach to Mitigate Security Interference

Resource sharing in public clouds enables a new avenue of security threats that are mounted across the virtual machine boundary. In this chapter, we describe a host-centric approach to mitigate two cross-VM attacks—a performance degradation attack and an information leakage attack—that abuse the interference on shared CPU cache. The design of our solution is based on a key observation made in Chapter 6. That is, to be effective, many cache-based cross-VM attacks rely on frequent VM preemption to intensify the interference caused by cache resource contention.

Our mitigation technique focuses on containing such frequent VM preemption. However, because preemptive CPU scheduling is designed to allow timely processing of I/O interrupts for latency-sensitive workloads, the challenge of this approach is to improve security without incurring undue performance overhead to the benign workloads that are latency-sensitive. To solve this problem, our design controls cache sharing with a partition-based approach that separates latency-bound and throughput-bound VMs and runs them on two disjointed groups of CPU cores. While our system does not completely eliminate the threats, testbed experiments show that it can significantly reduce the *effectiveness* of both target attacks that abuse the shared CPU cache. Importantly, this approach also *improves* the performance of latency-bound and throughput-bound workloads, instead of incurring undue overhead.

7.1 Related Work

Two general approach exist to mitigate the cross-VM performance degradation attacks and information leakage attacks (including both covert channels and side channels) that abuse the share resources. One targets the resources being abused, such as the CPU cache and memory bus, and the other targets the attacking mechanisms, which we explain later.

Most existing solutions take the first approach. DeepDive [58] migrates VMs to other physical machines if the CPU performance counters indicate that these VMs are causing abnormal resource contention. This is a general purpose method to avoid interference as long as there are performance counters available for monitoring the usage of the target resources. Wang *et al.* designed a hardware-based solution that either partitions the share cache lines or randomizes cache line sharing [79]. Moreover, the access to the shared resources can be regulated in software with modifications to the hypervisor. For example, Godfrey *et al.* proposed to flush the shared CPU cache between schedules to disallow information leakage [30]. StealthMem [45] provided guest VMs with new hypercalls to use private memory for security-sensitive operations. BusMonitor [67] traps the atomic instructions issued by guest VMs using shadow paging; these instructions are essential to the covert channel attacks that abuse the shared memory bus [85].

Alternatively, these cross-VM attacks can be mitigated by impairing the key mechanisms that enable the attacks in the first place regardless of the shared resources. Askarov *et al.* demonstrated that timing-based information leakage can be mitigated by regulating all of the timing events in the system [9]. Coppens *et al.* proposed to use a special compiler backend to eliminate timing behaviors [19]. XenPump [84] mitigates timing-based attacks by adding latencies to critical hypercalls. Our solution discussed in this chapter also falls into this category. It leverages the observation that many cache-based attacks rely on frequent VM preemption to be effective. Therefore, the goal of our design is to prevent excessive VM preemption to reduce conflicts of resource usage and to do so without significantly penalizing the performance of benign workloads. The solutions that target the shared resources and the solutions that target the attacking mechanisms are in fact complementary to each other. A qualitative comparison is presented in § 7.2.3.

7.2 Design

In this section, we present the design of a host-centric system that reduces the effectiveness of a performance degradation attack and an information leakage attack that abuse the shared CPU cache. The discussion starts with the design trade-offs in mitigating the security interference, which motivates the partition-based approach that is described next. To conclude the design section, we qualitatively compare our solution to the alternatives regarding their effectiveness and performance overhead.

7.2.1 Design Trade-offs

If the goal of our system is only to prevent frequent VM preemption, the solution is rather simple—we just need to adjust the preemption rate of the VM scheduler. Xen 4.2 introduced a scheduling parameter called `rate` that configures the minimum amount of time a VM can run before other VMs are allow to preempt it [88]. In other words, `rate` determines the maximum allowable preemption rate, which is precisely what is needed for our solution. In its default setting, guest VMs cannot be preempted more often than once every millisecond.

By adjusting the `rate` parameter, we can directly control the effectiveness of the cache-based cross-VM attacks that rely frequent VM preemption. For example, in the case of the performance degradation attack with the attacker VM and victim VM sharing the same CPU core (§ 6.1), if we set `rate` to be 5ms, then the attacker VM cannot evict the victim's contents from the shared cache more often than once very five milliseconds. Unfortunately, this native approach would render the latency-sensitive workloads non-responsive because they also rely on frequent VM preemption for timely interrupt handling. With `rate` being 5ms, latency-sensitive VMs also have to wait up to 5ms to process their pending interrupts if they are sharing processors with throughput-bound VMs. In other words, there exists a trade-off between the security of cache-sensitive workloads and the responsiveness of latency-bound workloads. Any particular configuration of the VM scheduling policy may not satisfy all workloads.

7.2.2 A Partition-Based Design

To mitigate the cache-based cross-VM attacks without undue performance overhead to benign workloads, our solution uses a *partition-based* approach. It classifies guest VMs into one of the two types—latency-bound or throughput-bound—according to the workloads running inside and schedules them on disjointed groups of CPU cores. Importantly, when allocating CPU cores for each group, we want to minimize the resources shared across groups. For example, given a four-core CPU with one L1 cache on each core and two L2 caches shared by two pairs of cores respectively, we can divide the cores into two groups so that the cores in different groups share no cache but the memory bus. This design is considered host-centric because the workload classification can be done in the host without any guest cooperation.

Note that in our workload classification scheme—latency-bound versus throughput-bound—there is no explicit security consideration. This is by design because it is difficult for cloud providers to reliably identify whether a guest VM is attacking its neighbors. The evaluation results in § 7.4 show that classifying VMs solely by their performance characteristics is sufficient to mitigate the two target cache-based cross-VM attacks that rely on frequent VM preemption. Meanwhile, because workloads of different performance characteristics now run in separate groups and have a little impact on each other across groups, we can keep the scheduler policy (e.g., the `rate` parameter) as is.

The remaining question is how to achieve such workload classification. Existing mechanisms either adopt a gray-box approach by explicitly monitoring I/O events [31, 44, 39] or require guest VM cooperation [53, 50, 89]. We use a lightweight host-centric design to classify workloads based how VMs consume CPU cycles. Specifically, we monitor the average burst length of CPU usage for each VM and classify a VM as latency-bound if its burst length is below a threshold, and the rest are throughput-bound. The intuition is straightforward: latency-bound VMs by definition do not consume much CPU time to process interrupts and generate responses before yielding to wait for the next I/O event.

Essentially, this partition-based design offers a mechanism to control cache sharing between incompatible workloads, like what Bobtail (Chapter 4) does for processor sharing,

but it does so at the host level without guest cooperation. Therefore, a throughput-bound VM is sensitive to the attacks but hard to be attacked because its neighboring VMs with shared cache are also by definition throughput-bound and use lots of CPU time before yielding or being preempted. On the other hand, a latency-bound VM is easy to be attacked but insensitive to the attacks because such VMs by definition do not use too many CPU cycles and are not susceptible to attacks when waiting for interrupts. The application of this argument to the two target attacks discussed in this chapter is as follows. Note that in either case, a attacker VM needs to be in the same scheduling group as the victim in order to share CPU cache.

Performance degradation attack If a victim VM is classified as throughput-bound, its workload is sensitive to cache misses. However, because its neighboring VMs in the same group are also throughput-bound, they cannot preempt the victim as often as before the partitioning; otherwise they cannot stay in this group. As a result, the efficiency of the attack would be greatly reduced due to the lower *effective* preemption rate. On the other hand, if classified as latency-bound, the victim VM is equally vulnerable as when there is no partitioning, i.e., our partition-based design offers neither advantage nor disadvantage over Xen's default scheduling mechanism for latency-bound VMs. However, by definition, the CPU usage of such VMs is less bursty than throughput-bound VMs. Therefore, when they do not need to use physical CPUs, they are not subject to the attack either.

Information leakage attack The story for information leakage attacks is more complicated. In theory, both cache-based covert channel (§ 6.2) and side channel attacks [97] rely on frequent VM preemption to probe the shared cache for leaked information, and we can control their effectiveness with partitioning just like the performance degradation attack. However, besides preemption, a VM can also yield its assigned physical CPU *voluntarily*. As a result, in the case of covert channel attacks, for which both the source and destination of the information channel are controlled by the attacker, the two VMs can collude with each other to yield physical CPUs in lockstep and avoid the need of preemption. In other words, yielding physical CPUs voluntarily can defeat our mitigation mechanism and allows

colluding VMs to stay in the latency-bound group with synchronized cache usage, but it is only possible when both ends of the information channel can collude.

On the other hand, our partition-based approach can reduce the effectiveness of the cache-based side channel attack. As discussed in § 6.2, it is essentially a special case of the covert channel attack as it has no control over the source VM of the leaking channel. In this scenario, preemption rate becomes relevant again. Specifically, if a victim (source) VM is classified as throughput-bound, other VMs in the same scheduling group may only probe the shared CPU cache at a slower speed in order to stay in the same group with shared cache. Therefore, less information can be leaked per unit of time. On the other hand, for victim VMs that are latency-bound, we may only rely on the default maximum allowable preemption rate (once per millisecond) to contain the information leakage. But at the same time, such VMs do not leak information when they are blocked for external events.

7.2.3 Design Space Comparison

The key insight of our partition-based approach is to mitigate the cross-VM attacks by impairing their attacking mechanisms. In the case of the cache-based performance degradation attack and side channel attack, the goal is to reduce the VM preemption rate without undue harm to the performance of benign workloads. Compared to other defense mechanisms, our approach has a simpler design and incurs less performance overhead. In fact, as shown in § 7.4, our approach *improves* performance of benign workloads as a side effect. Meanwhile, the disadvantage of our approach is that it mitigates fewer types of attacks (e.g., it is not effective against the information leakage attack that abuses the shared memory bus [85]) than the existing solutions, and it only reduces the negative impact of the attacks instead of eliminating them.

The idea of partition-based scheduling is not new by itself. For example, a similar technique has been proposed to improve I/O performance by separating VMs running I/O operations from the ones running compute-intensive jobs and configuring them with different scheduling strategies; as a trade-off, the performance of compute-intensive applications is sacrificed [39]. In comparison, the security and performance benefits of our design are de-

rived from two assumptions regarding workload characteristics and VM scheduling mode, respectively. We make these assumptions based on our observations and experience in running experiments in public clouds:

Workload characteristics The existing methods that make a distinction between I/O-bound workloads and CPU-bound workloads usually assume that both I/O-bound workloads *and* CPU-bound workloads run concurrently in the target VM [31, 44, 39, 91, 90]. In this scenario, Xen's default credit scheduler would not `BOOST` the target VM because its CPU-bound workloads consume lots of CPU cycles, even though `BOOST`ing is essential for I/O-bound workloads to be responsive. This is a reasonable assumption because virtualization is often used for consolidating various types of workloads into the same physical machine. However, we observe that this assumption can be relaxed in the context of public clouds, whose programming and pricing models allow developers to rent VM instances of different size based on their actual resource requirements. For example, it is possible to rent a VM of smaller size (e.g., micro and small instances in EC2) at a lower price to run *one workload per VM*. Under this revised assumption, VM classification becomes easier because in a given period of time the workload in the target VM should only exhibit either latency-bound or throughput-bound characteristics, but not both.

Scheduling mode By default, CPU schedulers like Xen's credit scheduler use work-conserving mode to improve CPU utilization. That is, for a set of VMs sharing CPU cores, if they all run CPU-bound workloads, each VM gets a fair share of CPU cycles based on their relative weights. Meanwhile, if some of these VMs consume less CPU time than their fair share, the excess CPU cycles can be reallocated to the other VMs that need them. However, public clouds like Amazon EC2 usually employ a non-work-conserving mode to schedule various resources [56]. Therefore, no VM can use more than its fair share of physical CPU cores even if there are excess cycles. This choice allows VMs to perform more predictably than in work-conserving mode—a VM cannot run faster than what is advertised despite its neighbor's behavior. On the other hand, the lost CPU utilization is not of a concern for cloud service providers because they charge customers by the amount of resources

they rent not by the amount they actually use. As shown in § 7.4.2 this observation leads to an improved performance for CPU-bound workloads, while existing partition-based approaches conclude the opposite [39].

To sum up, these two assumptions not only simplify the design and implementation (§ 7.3) of VM classification, but also help improve both the security *and* performance (§ 7.4) of the partitioned workloads.

7.3 Implementation

To mitigate the cache-based performance degradation attack and side channel attack, our system consists of two major components: a resource management module to divide physical CPU cores into two separate groups and a VM classification module to assign guest VMs to different scheduling groups according to their workload characteristics. Our current implementation is based on Xen 4.2.1 and Linux kernel 3.6.6.

The scheduling related features introduced in Xen 4.2 make the implementation of the resource management module straightforward. Specifically, `cpupool` [87] allows administrators to group CPU cores and apply different scheduling algorithms and parameters to them with userspace tools available in dom0. As a result, to implement the resource management module, we only need to determine for each physical machine which processors should be allocated to latency-bound VMs and which ones should be allocated to throughput-bound VMs (besides the ones used by dom0), and then map each group to a `cpupool`. The key requirement for the allocation is to minimize the sharing of processor architecture resources (e.g., various levels of CPU caches) across `cpupools`. In our current implementation, such allocation is static for a given model of physical machines. To deploy this system in a production cloud, the allocation may become dynamic by taking into account the cloud-wide resource demand for latency-bound VMs and throughput-bound VMs. Due to a lack of access to production traces, we leave this problem for future work.

In addition, we implement the workload classification module by instrumenting Xen's VM scheduler layer. According to our design, we use the average burst length of CPU usage for each VM as the classification criteria. More precisely, we can only obtain such statistics

for virtual CPUs (VCPUs), of which a guest VM may have one or more. However, our current implementation targets the VMs that have only one VCPU (like EC2 small instances) so that we can hold the assumption of one workload per VM. Specifically, we instrument the generic scheduler layer of the Xen hypervisor so that the same functionality can also be applied to scheduling algorithms other than the credit scheduler. To do so, we record the runtime of a VCPU when its status changes from `running` to `blocked` and use shared memory pages to make such records available to user space tools running in dom0. Note that a VCPU may switch from `running` to `runnable` if preempted by the scheduler before changing to `blocked`. In this case, we do not consider it as a complete record because from the workload's perspective, its burst of CPU usage will continue once the corresponding VM is scheduled back. To calculate the average burst length, we need to set a sampling rate for the instrumentation in Xen's scheduler layer. In our current implementation, we record a VM's runtime every time it changes status and report the average length once per second.

Finally, the resource management module integrates with the workload classification module using a monitoring process in dom0. It collects VCPU statistics via the shared memory pages and assigns each VM to the right group according to the classification criteria. Note that because VMs may change their workload types from time to time, our static `cpupool` allocation scheme may result in resource imbalance on individual physical machines. Using live migration may solve this problem, but we consider it as future work.

7.4 Evaluation

In this section, we evaluate our partition-based VM scheduling scheme using controlled experiments on a testbed. The experiments demonstrate both the effectiveness of our mitigation technique for cache-base cross-VM attacks and its performance impact.

Testbed setup The testbed we use in this chapter is similar to that in § 5.3. It consists of a Cisco Catalyst 2970 switch connecting several physical machines, all of which have Linux 3.6.6 and Xen 4.2.1 installed. In our experiments, we instrument Xen's VM scheduler to collect the statistics for workload classification and leave the Linux kernel unmodified.

Parameters There are two key parameters for the system: the threshold of average CPU burst length for workload classification and the sampling rate for collecting CPU usage statistics. The optimal values of the these parameters are workload dependent and should be derived from the measurements of the production traces. Because only synthetic workloads are used for the evaluation, we apply 5ms for the threshold of workload classification as our best. In addition, we keep the sampling rate once per second. The impact of these values is then discussed in § 7.4.2.

7.4.1 Attack Mitigation

We demonstrate the effectiveness of our mitigation technique in three scenarios:

1. The attacker VM and the victim VM are classified into different scheduling groups.
2. Both the attacker VM and the victim VM are classified as throughput-bound.
3. Both the attacker VM and the victim VM are classified as latency-bound.

7.4.1.1 Performance Degradation Attack

Applying the attack described in § 6.1 directly results in the first scenario. The victim VM scans a memory buffer repeatedly so it is clearly throughput-bound. Because the attacker VM sleeps between scanning operations, regardless of the length of its sleep period, this VM is classified as latency-bound because its average CPU burst length, which corresponds to a full scan of the memory buffer (same size as the L2 cache), is less than the 5ms threshold. As a result, the attacker VM and the victim VM are classified into two scheduling groups that do not share L2 cache, which is equivalent to running these VMs on different CPU packages. As shown in § 6.1.2, the resulting effectiveness of the attack is no more than 13.8% increase in the runtime of the victim workload.

For the second scenario, we have to modify the attacker workload to make it enter and stay in the throughput-bound group as the victim. Given the 5ms cutoff, we let it scan the allocated memory buffer three times before sleeping. Intuitively, the effectiveness of the attack is sacrificed as it spends more time on repeating itself without evicting the victim's cache contents.

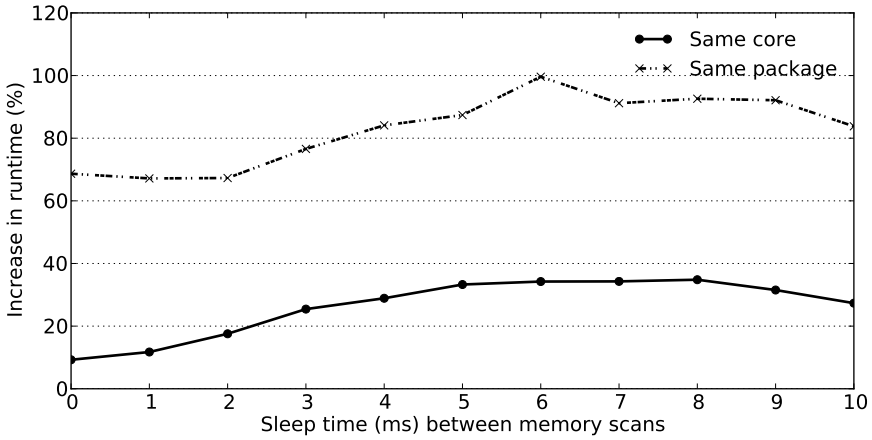


Figure 7.1: The effectiveness of the cache-based performance degradation attack after restricting both the attacker VM and the victim VM in the throughput-bound group with partition-based scheduling.

Figure 7.1 supports this intuition. Compared with Figure 6.1 in the characterization study in § 6.2, the effectiveness of this attack, as depicted by the maximum runtime increase, is reduced from 77.9% to 34.8% for the "same core" case and from 129.5% to 99.6% for the "same package" case.

Finally, for the third scenario for which both the attacker VM and the victim VM are in the latency-bound group, our partition-based scheduling scheme offers neither advantage nor disadvantage over Xen's default scheduler. Therefore, its evaluation is omitted.

7.4.1.2 Information Leakage Attack

As discussed in § 7.2.2, the partition-based mitigation technique is effective against the information leakage attack using the cache-based side channel, but not the covert channel. To demonstrate its effectiveness, we also follow the three scheduling scenarios as for the performance degradation attack.

The workloads we use for the demonstration is based on the attack designed by Zhang *et al.* [97]. It consists of a measurement stage and an analysis stage. The analysis stage uses the data obtained in the measurement stage offline to uncover the bits in the private key used and leaked by a vulnerable software package running in the victim VM. Therefore, this second stage is irrelevant to our study, and we use the effectiveness of the measurement stage to represent the effectiveness of the attack.

Specifically, Zhang *et al.* report that they performed $300,000,000$ measurement trials in work-conserving mode and $1,900,000,000$ trials in non-work-conserving mode to gather enough data for further analysis—the first experiment lasted about six hours and the second lasted about 45 hours [97]. In our evaluation, we show how much time we would need to conduct the same amount of measurement trials as the metric of the attacking effectiveness after applying workload partitioning. Note that for this particular attack, the L1 cache is probed for leaked information. It means on our testbed, the attacker VM and the victim VM must share the same CPU core.

To begin with, if we apply their measurement methodology directly, the victim VM is classified as throughput-bound as it is required to conduct crypto operations repeatedly with the same private key, and the attacker VM appears latency-bound because it repeats a measure-then-idle cycle with the measuring operation taking less than 500 microseconds. Therefore, the victim VM does not share CPU cache with the attacker VM, and no leaked information can be learned as before. To simplify our experiments, we use a synthetic victim workload that uses memory scanning operations instead of crypto operations to spin the CPU because we do not need any useful data to perform the analysis stage.

In the scenario of co-scheduling as throughput-bound VMs, we need to slow down the measurement operations of the attack VM to make each period last at least 5ms before idling; otherwise it cannot be classified as throughput-bound. The consequence of this modification is that the attacker VM can only effectively probe the shared L1 cache once every 5ms at the best. As a result, in the non-work-conserving mode, it would at least take us over $2,600$ hours to perform $1,900,000,000$ trials.

Finally, to construct the third scenario with the two VMs in the same latency-bound group, we modify the victim workload to insert 1ms sleep times between sets of memory scanning operations. Each set of the memory scanning operations is adjusted to last a little less than 5ms so that the victim VM appears as latency-bound to the workload classifier. In this case, the Xen's default 1ms rate limit mechanism can help contain the effectiveness of the attacker's probing operations because the attacker VM cannot preempt the victim VM more often than once every 1ms under rate limiting. As a result, to perform $1,900,000,000$ trials of effective cache probing, it would take at least 528 hours.

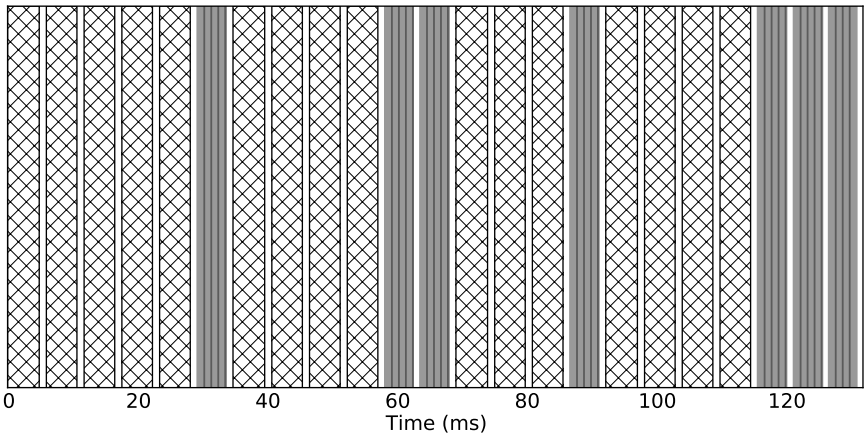


Figure 7.2: An excerpt of the execution pattern of the victim VM running a latency-bound workload suffering from the side channel attack.

Interestingly, in this third scenario, there two other conditions that further limit the effectiveness of cache probing. We use Figure 7.2 to explain this observation. This figure shows an excerpt of the execution pattern of the victim VM. It is generated with Xen's VM scheduling trace for about 130ms. There are three different types of rectangles in the figure, each of which represents an execution state. The width of the rectangles corresponds to the length of the execution time for each state. Among the three types of rectangles, the thin rectangles filled with the gray color are the expected 1ms execution periods. Specifically, the victim VM runs for a little over 1ms then gets preempted for a short while by the attack VM to probe the shared L1 cache. This pattern repeats until the victim finishes a set of memory scanning operation and sleeps. However, there are two other types of rectangles. The thin empty ones represent the 1ms sleep times between memory scanning operations. During this periods, the attack VM cannot learn any useful information because the victim is not running. In addition, the thick rectangles with the grid pattern represent the continuous execution of the victim without preemption. This state happens when the attacker VM does not have a high enough priority to preempt the victim VM (e.g., the attacker VM runs out of credits). As a result, during these periods, the attacker VM cannot execute roughly once very 5ms, which is similar to the second scenario with the two VMs in the throughput-bound group. With these two conditions combined, the time for the attacker to perform $1,900,000,000$ trials of effective cache probing is expected to be significantly higher than 528 hours.

Scenarios	Runtime (throughput-bound)	99th Latency	99.9th Latency
Baseline	54.45s	1.44ms	13.26ms
Partitioned	52.69s	0.45ms	0.48ms
Improvement	3.23%	68.75%	96.38%

Table 7.1: The performance impact of the partition-based scheduling scheme for benign workloads.

To sum up, regardless of which scheduling group the victim VM belongs, the expected effectiveness of the measurement stage for the side channel attack is significantly reduced by our partition-based scheduling scheme.

7.4.2 Performance Impact

In this subsection, we demonstrate the performance impact of our partition-based scheduling scheme. We allocate four VMs on the same physical machine to share two CPU cores. Each VM can use up to 50% of one physical core. Our experiments use a latency-bound workload that runs a `Thrift` RPC server [71] and a throughput-bound workload that repeats a cycle of scanning a L2-cache-sized buffer 100 times and sleeping for 1ms. These two workloads are designed to be partitioned into separate scheduling groups.

The evaluation includes two scenarios that co-schedule two latency-bound VMs and two throughput-bound VMs on the two shared CPU cores. We demonstrate the performance impact of our approach by comparing the runtime of the throughput-bound workload and the tail latency (the 99th and 99.9th percentiles) of the latency-bound workload. For both cases, the shared CPU cores reside in different packages so that they do not share L2 cache. Note that the overhead of workload classification is negligible because it only happens at most once per second in our current implementation.

In the first scenario, the four guest VMs are allowed to float across the shared CPU cores. In other words, it is up to the default credit scheduler to determine how to map guest VMs to available CPU cores. This scenario represents the baseline. In the second scenario, the VMs are partitioned by their workloads—the two throughput-bound VMs are scheduled on one core and the latency-bound VMs are scheduled on the other. The two groups of VMs are not allowed to migrate across cores.

Table 7.1 shows the performance comparison between the two scenarios. Our partition-based scheduling brings improved performance across all three metrics, instead of introducing undue overhead. It improves tail network latency because after partitioning, latency-bound VMs only need to compete for the CPU core with each other. By definition, latency-bound workloads do not use much CPU time between schedules. Therefore, they do not delay the interrupt processing of each other. This is essentially the same reason why Bobtail (Chapter 4) works. The difference is that the solution described in this chapter is host-centric and partitions at the CPU core level, while Bobtail is guest-centric partitions at the physical machine level.

In addition, the runtime of throughput-bound workloads also improves because by avoiding sharing CPU cores with latency-bound workloads, the throughput-bound workloads experience fewer cache misses and context switches. However, their improvement shown in Table 7.1 is rather marginal—merely 3.23%. After an investigation, we find that the throughput-bound VMs are scheduled on the CPU core that shares L2 cache with the core that runs dom0. Recall that Xen's device driver model uses dom0 to handle I/O interrupts and send virtual interrupts to other guest VMs. As a result, the throughput-bound VMs still experience cache misses caused by the activity of dom0, and we only get a marginal improvement in their runtime. To verify this speculation, we switch the CPU cores for the two types of VMs and schedule the throughput-bound VMs in the package other than the one that runs dom0. Not surprisingly, the same experiment with this new configuration now yields a 16.64% improvement in the runtime of the throughput-bound VMs.

7.4.3 Parameterization

In this subsection, we discuss the impact of the parameters for the partition-based scheduling scheme. There are two parameters in the system—the threshold of average CPU burst length for workload classification, which is 5ms in our experiments, and the sampling rate for collecting CPU usage statistics, for which we use once per second by default.

The value of the classification threshold can shape the behaviors of malicious workloads. Take the performance degradation attacks as an example. The attacker VM must

increase its average CPU burst length to exceed the threshold value, in order to be classified as throughput-bound. In other words, a larger threshold translates to a larger burst of CPU usage for workloads in the throughput-bound group. Benign workloads are therefore better protected in this group. On the flip side, any workloads with their CPU burst length below the threshold are classified into the latency-bound group, which does not benefit from the partition-based scheduling. In short, a larger threshold value offers *better* protection for *fewer* workloads. This conclusion is also applicable to the information leakage attack.

The sampling rate determines how long a VM can stay in a scheduling group before reclassification. A malicious VM can fake its workload type and stay undetected for every other sampling period. For example, with a one-second sampling rate, a latency-bound VM can temporally make itself throughput-bound to get classified in the wrong group for one second and attack its target VM with frequent preemption. But it will be sent back to the latency-bound group the next second. As a result, for security reasons, we should favor a large sampling rate as long as the resulting performance overhead is acceptable.

7.5 Summary

In this chapter, we mitigate the security interference between virtual machines by reducing the effectiveness of two cross-VM attacks—a performance degradation attack and an information leakage attack—that abuse the shared CPU cache. Our solution is a partition-based VM scheduling system with a host-centric design. Its intuition is to reduce frequent VM preemption, which is critical to the effectiveness of the target attacks, without negatively affecting the performance of benign workloads. This system works by scheduling latency-bound VMs and throughput-bound VMs separately on disjointed groups of processors. The processor groups are allocated with minimized resource sharing across groups in order to restrict cache sharing to only compatible workloads. Evaluation results show that this scheduling scheme not only reduces the effectiveness of the target attacks but also improves the performance of benign workloads due to controlled resource sharing, instead of incurring undue overhead.

CHAPTER 8

Conclusion

The paradigm of public cloud computing provides developers with the ability to build Internet-scale applications without an upfront hardware investment, but it also presents new challenges in ensuring the performance and security of the data center infrastructure. The value of this dissertation is to study the interference between guest virtual machines (VMs) in public clouds and design mitigation strategies. For performance problems, it characterizes the impact of the interference on inter-VM network latency using live measurements in a real public cloud, and it uncovers the root cause of the negative impact with controlled experiments on a local testbed. Two methods are proposed to improve the inter-VM network latency: a guest-centric solution is designed to exploit the properties of application workloads to avoid the interference without any support from the underlying host infrastructure; a host-centric solution is designed to adapt the scheduling policies for the shared resources that cause the interference without guest cooperation. For security problems, this dissertation characterizes two cross-VM attacks that abuse the shared CPU cache. To mitigate such security interference, it designs a partition-based VM scheduling system, from the perspective of the host infrastructure, to reduce the effectiveness of these cache-based attacks without introducing undue harm to the performance of benign workloads. The next few sections summarize some of the key insights this dissertation has produced, discuss the limitations, and suggest future work.

8.1 Insights

The key insight behind the proposed solutions is that by identifying the shared resources that cause the interference and exploiting the properties of the workloads that share these resources with adapted scheduling policies, public cloud services can reduce conflicts of resource usage between guests and hence mitigate their interference.

Specifically, this dissertation has provided the insight that not all resource sharing is bad. In public cloud environments, hardware resources are inevitably shared, and the performance and security of guest applications are often negatively impacted by such sharing. However, this dissertation has shown that if resource sharing is confined to only compatible workloads, its negative impact on the performance and security of guest VMs can be significantly reduced. While the exact definition of workload compatibility may depend on the shared resources in question, generally speaking, the VMs that have a similar resource access pattern usually play well with each other.

In addition, the work of avoiding multiple latency traps in virtualized data center infrastructure has provided the insight into the trade-offs between network throughput and latency for cloud applications. We have shown that many latency problems are caused by the design choices made to optimize for throughput, instead of by fundamental limitations, like the speed of light. As a result, when designing user-facing applications and the infrastructure that supports such applications, we can reassess the resource scheduling policies that balance throughput and latency and develop novel techniques to significantly reduce network latency without negatively impacting throughput.

Finally, the techniques developed to mitigate the interference of shared processors between guest virtual machines has also provided the insight that by considering the unique properties of public cloud computing, such as the use of non-work-conserving scheduling, and exploiting the performance characteristics of guest workloads, we can deal with security problems with lightweight mechanisms without trading-off the performance of benign workloads. While there exists a semantic gap between guest VMs and the host infrastructure, many important properties of guest workloads can still be leveraged to contain the negative impact of resource sharing on security and performance. In short, the paradigm of

public cloud computing makes the problem of resource management more challenging than private or dedicated data centers, but it also offers new opportunities to solve the problem in unique ways.

8.2 Limitations

The techniques developed in this dissertation are evaluated using synthetic workloads and simulations. While such experimentation techniques allow fine-grained control of variables and enable in-depth exploration of system design choices, they may not be sufficient to reflect the properties of the systems running production workloads. Specifically, compared with production workloads, the scale and complexities of synthetic workloads may not be very representative. As a result, we may have overlooked important factors that affect system behaviors in practice.

In addition, this dissertation has discussed techniques that mitigate the performance and security interference between guest virtual machines instead of eliminating them all together. In other words, the resource scheduling mechanisms discussed in this dissertation do not improve virtual machine isolation directly but rather contain the negative impact of imperfect isolation. This limitation is the result of the trade-off between complexity and effectiveness. While cloud applications that require the absence of performance and security interference certainly exist, improving the effectiveness often leads to increased complexity in system design. Therefore, the solutions proposed in this dissertation aim to strike a balance between the two and offer a reasonable trade-off for common applications.

Finally, regardless of the effort to strike such balance, the solutions offered by this dissertation inevitably increase the complexity of cloud infrastructure. When considering malicious cloud guests, it means these solutions also provide new avenues to attack or game the system. For example, the two-level prioritization mechanism to reduce the queueing delay found on data center switches could be exploited by greedy guests; they could gain a higher priority by breaking down large flows into smaller ones using colluding VMs. However, such problems may only become evident when the systems are deployed in production environments, which is an important problem to be addressed in future work.

8.3 Future Work

One avenue of work unexplored in this dissertation is the mitigation of the interference caused by other shared resources, such as disks and GPUs. The diversity of cloud applications stresses various aspects of resource scheduling in the public cloud infrastructure. Therefore, a categorization of shared hardware resources with respect to application performance and security in the context of public clouds is required. Both the application of existing scheduling techniques to new resources and the exploration of new techniques are worth future effort.

Another avenue for future work is to apply similar studies to different cloud service providers and virtualization hypervisors (other than EC2 and Xen). Many problems and solutions explored in this dissertation can be generalized to other infrastructure vendors. However, new public cloud services, especially the ones using different hypervisor software, also offer opportunities to discover new challenges. In addition, like [52], a horizontal comparison of popular public cloud services for performance and security interference would be valuable to both the developer and the research communities.

Finally, to address a key limitation of this dissertation, an important future work is to deploy the proposed solutions in production cloud environments using production workloads. Doing so is challenging in several aspects. In the case of Bobtail discussed in Chapter 4, there are already extensive experiments conducted in real EC2 clouds; what is left is to apply them to production workloads. However, other solutions proposed in this dissertation require modification to the underlying cloud infrastructure, such as the hypervisor and host operating system kernel, whose deployment can be challenging for commercial public clouds. An alternative is to build a larger scale testbed, e.g., using a few hundred physical machines, and then to deploy the systems described in this dissertation with production workloads or traces. Regardless, evaluating the proposed solutions using a more realistic setup should be an integral part of a future research agenda.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Christopher Abad. IP Checksum Covert Channels and Selected Hash Collision. Technical report, 2010.

[2] Jeongseob Ahn, Changdae Kim, Jaeung Han, Young ri Choi, and Jaehyuk Huh. Dynamic Virtual Machine Scheduling in Clouds for Architectural Shared Resources. In *Proceedings of the 4th USENIX conference on Hot Topics in Cloud Computing (HotCloud'12)*, Boston, MA, USA, June 2012.

[3] Kamran Ahsan and Deepa Kundur. Practical Data Hiding in TCP/IP. In *Proceedings of the 9th workshop on Multimedia & Security, (MM&Sec'02)*, Dallas, TX, USA, September 2002.

[4] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of the ACM SIGCOMM 2008 conference (SIGCOMM'08)*, Seattle, WA, USA, August 2008.

[5] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data Center TCP (DCTCP). In *Proceedings of the ACM SIGCOMM 2010 conference (SIGCOMM'10)*, New Delhi, India, August 2010.

[6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation (NSDI'12)*, San Jose, CA, USA, April 2012.

[7] Mohammad Alizadeh, Shuang Yang, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. Deconstructing Datacenter Packet Transport. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks (HOTNETS'12)*, Redmond, WA, USA, October 2012.

[8] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing Router Buffers. In *Proceedings of the ACM SIGCOMM 2004 conference (SIGCOMM'04)*, Portland, OR, USA, August 2004.

[9] Aslan Askarov, Danfeng Zhang, and Andrew Myers. Predictive Black-box Mitigation of Timing Channels. In *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS'10)*, Chicago, IL, USA, October 2010.

[10] Hitesh Ballani, Keon Jang, Thomas Karagiannis, Changhoon Kim, Dinan Gunawardena, and Greg O'Shea. Chatty Tenants and the Cloud Network Sharing Problem. In *Proceedings of the 10th USENIX Symposium on Networked System Design and Implementation (NSDI'13)*, Lombard, IL, April 2013.

[11] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, Bolton Landing, NY, USA, October 2003.

[12] Sean K. Barker and Prashant Shenoy. Empirical Evaluation of Latency-sensitive Application Performance in the Cloud. In *Proceedings of the 1st annual ACM SIGMM conference on Multimedia systems (MMSys'10)*, Scottsdale, AZ, USA, February 2010.

[13] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 2010 Internet Measurement Conference (IMC'10)*, Melbourne, Australia, November 2010.

[14] K.F. Bowdena, I.R. MacCalluma, and S.P. Patience. A Magnetic Tape Database for a Real-Time Medical Information System. *Computers in Biology and Medicine*, 5, September 1975.

[15] Serdar Cabuk, Carla E. Brodley, and Clay Shields. IP Covert Timing Channels: Design and Detection. In *Proceedings of the 11th ACM Conference on Computer and Communications Security (CCS'04)*, Washington, DC, USA, October 2004.

[16] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, Seattle, WA, USA, November 2002.

[17] Peter Chen and Brian Noble. When Virtual Is Better Than Real. In *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems (HOTOS'01)*, Washington, DC, USA, May 2001.

[18] Luwei Cheng and Cho-Li Wang. vBalance: Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines. In *Proceedings of ACM Symposium on Cloud Computing 2012 (SoCC'12)*, San Jose, CA, USA, October 2012.

[19] Bart Coppens, Ingrid Verbauwhede, Bjorn De Sutter, and Koen De Bosschere. Practical Mitigations for Timing-Based Side-Channel Attacks on Modern x86 Processors. In *Proceedings of the 30th IEEE Symposium on Security & Privacy (S&P'09)*, Oakland, CA, USA, May 2009.

[20] Don Coppersmith. Small Solutions to Polynomial Equations, and Low Exponent RSA Vulnerabilities. *Journal of Cryptology*, 10:233–260, 1997. 10.1007/s001459900030.

[21] Jeff Dean. Designs, Lessons and Advice from Building Large Distributed Systems. In *Proceedings of the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)*, Big Sky, MT, USA, October 2009. Keynote speech.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation (OSDI'04)*, San Francisco, CA, USA, March 2004.

[23] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)*, Stevenson, WA, USA, October 2007.

[24] Digital Forensics Association. *The Leaking Vault Five Years of Data Breaches*, July 2010.

[25] George W. Dunlap. Scheduler Development Update. In *Xen Summit Asia 2009*, Shanghai, China, November 2009.

[26] Amazon EC2. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://aws.amazon.com/message/65648/`.

[27] Benjamin Farley, Venkatanathan Varadarajan, Kevin Bowers, Ari Juels, Thomas Ristenpart, and Michael Swift. More for Your Money: Exploiting Performance Heterogeneity in Public Clouds. In *Proceedings of ACM Symposium on Cloud Computing 2012 (SoCC'12)*, San Jose, CA, USA, October 2012.

[28] Simson Garfinkel. *Architects of the Information Society: Thirty-Five Years of the Laboratory for Computer Science at MIT*. MIT Press, 1999.

[29] John Giffin, Rachel Greenstadt, Peter Litwack, and Richard Tibbetts. Covert Messaging Through TCP Timestamps. In *Proceedings of the 2nd Workshop on Privacy Enhancing Technologies (PET'02)*, San Francisco, CA, USA, April 2002.

[30] Michael Godfrey and Mohammad Zulkernine. A Server-Side Solution to Cache-Based Side-Channel Attacks in the Cloud. In *Proceedings of the 6th IEEE International Conference on Cloud Computing (CLOUD'13)*, Santa Clara, CA, USA, June 2013.

[31] Sriram Govindan, Arjun R. Nath, Amitayu Das, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Xen and Co.: Communication-Aware CPU Scheduling for Consolidated Xen-based Hosting Platforms. In *Proceedings of the 3rd international conference on Virtual execution environments (VEE'07)*, San Diego, CA, 2007, June 2007.

[32] Liang Guo and Ibrahim Matta. The War Between Mice and Elephants. In *Proceedings of the Ninth International Conference on Network Protocols (ICNP'01)*, Riverside, CA, USA, November 2001.

[33] T. J. Hacker, B. D. Noble, and B. D. Athey. Improving Throughput and Maintaining Fairness Using Parallel TCP. In *Proceedings of the 23rd conference on Information communications (INFOCOM'04)*, Hong Kong, China, March 2004.

[34] Eric Hammond. Matching EC2 Availability Zones Across AWS Accounts. `http://http://alestic.com/2009/07/\ec2-availability-zones`.

[35] Mor Harchol-Balter, Bianca Schroeder, Nikhil Bansal, and Mukesh Agrawal. Size-Based Scheduling to Improve Web Performance. *ACM Transactions on Computer Systems*, 21(2):207–233, May 2003.

[36] John L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34, September 2006.

[37] Tom Herbert. bql: Byte Queue Limits. `http://lwn.net/Articles/454378/`.

[38] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing Flows Quickly with Preemptive Scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)*, Helsinki, Finland, August 2012.

[39] Yanyan Hu, Xiang Long, Jiong Zhang, Jue He, and Li Xia. I/O Scheduling Model of Virtual Machine Based on Multi-core Dynamic Partitioning. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*, hicago, IL, USA, June 2010.

[40] Information Sciences Institute. RFC 793: Transmission Control Protocol, 1981. http://www.ietf.org/rfc/rfc793.txt.

[41] Intel LAN Access Division. Intel VMDq Technology. Technical report, Intel, March 2008.

[42] Vimalkumar Jeyakumar, Mohammad Alizadeh, David Mazières, Balaji Prabhakar, Changhoon Kim, and Albert Greenberg. EyeQ: Practical Network Performance Isolation at the Edge. In *Proceedings of the 10th USENIX Symposium on Networked System Design and Implementation (NSDI'13)*, Lombard, IL, USA, April 2013.

[43] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B. Lee. NoHype: Virtualized Cloud Infrastructure without the Virtualization. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010.

[44] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, and Joowon Lee. Task-aware Virtual Machine Scheduling for I/O Performance. In *Proceedings of the 5th international conference on virtual execution environments (VEE'09)*, Washington, DC, USA, March 2009.

[45] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. STEALTHMEM: System-Level Protection Against Cache-Based Side Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium (SECURITY'12)*, Bellevue, WA, USA, August 2012.

[46] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology (CRYPTO'96)*, London, UK, August 1996.

[47] Younggyun Koh, Rob C. Knauerhase, Paul Brett, Mic Bowman, Zhihua Wen, and Calton Pu. An Analysis of Performance Interference Effects in Virtual Environments. In *Proceedings of the 2007 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'07)*, San Jose, CA, USA, April 2007.

[48] Craig Labovitz. How Big is Amazon's Cloud? `http://www.deepfield.net/2012/04/how-big-is-amazons-cloud/`.

[49] Butler W. Lampson. A Note on the Confinement Problem. *Communication of ACM*, 16:613–615, October 1973.

[50] Min Lee, A. S. Krishnakumar, P. Krishnan, Navjot Singh, and Shalini Yajnik. Supporting Soft Real-Time Tasks in the Xen Hypervisor. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'10)*, Pittsburgh, PA, USA, March 2010.

[51] Lydia Leong, Douglas Toombs, Bob Gill, Gregor Petri, and Tiny Haynes. Magic Quadrant for Cloud Infrastructure as a Service. *Gartner*, October 2012.

[52] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 2010 Internet Measurement Conference (IMC'10)*, Melbourne, Australia, November 2010.

[53] Bin Lin and Peter A. Dinda. VSched: Mixing Batch And Interactive Virtual Machines Using Periodic Real-time Scheduling. In *Proceedings of the 2005 ACM/IEEE conference on Supercomputing (SC'05)*, Seattle, WA, November 2005.

[54] Amazon Web Services LLC. Amazon Elastic Compute Cloud (EC2). `http://aws.amazon.com/ec2/`.

[55] Yiduo Mei, Ling Liu, Xing Pu, and Sankaran Sivathanu. Performance Measurements and Analysis of Network I/O Applications in Virtualized Cloud. In *Proceedings of the 3rd IEEE International Conference on Cloud Computing (CLOUD'10)*, Miami, FL, USA, June 2010.

[56] Jeffrey C. Mogul and Lucian Popa. What We Talk About When We Talk About Cloud Network Performance. *ACM SIGCOMM Computer Communication Review*, 42(5):44–48, October 2012.

[57] Kathleen Nichols and Van Jacobson. Controlling Queue Delay. *Queue*, 10(5):20:20–20:34, May 2012.

[58] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently Identifying and Managing Performance Interference in Virtualized Environments. In *Proceedings of the USENIX 2013 Annual Technical Conference (ATC'13)*, San Jose, CA, USA, June 2013.

[59] NS-3. http://www.nsnam.org/.

[60] Keisuke Okamura and Yoshihiro Oyama. Load-Based Covert Channels Between Xen Virtual Machines. In *Proceedings of the 25th Symposium on Applied Computing (SAC'10)*, Sierre, Switzerland, March 2010.

[61] Diego Ongaro, Alan L. Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *Proceedings of the 2008 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'08)*, Washington, DC, USA, March 2008.

[62] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. Exploiting Hardware Heterogeneity within the Same Instance Type of Amazon EC2. In *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'12)*, Boston, MA, USA, June 2012.

[63] OW2. RUBIS. http://rubis.ow2.org/.

[64] Colin Percival. Cache Missing For Fun And Profit. In *Proceedings of BSDCan 2005*, Ottawa, Canada, May 2005.

[65] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)*, Helsinki, Finland, August 2012.

[66] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud! Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS'09)*, Chicago, IL, USA, November 2009.

[67] Brendan Saltaformaggio, Dongyan Xu, and Xiangyu Zhang. BusMonitor: A Hypervisor-Based Solution for Memory Bus Covert Channels. In *Proceedings of the 2013 European Workshop on System Security (EUROSEC'13)*, Prague, Czech Republic, April 2013.

[68] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime Measurements in the Cloud: Observing, Analyzing, and Reducing Variance. In *Proceedings of the 36th International Conference on Very Large Data Bases (VLDB'10)*, Singapore, September 2010.

[69] Linus E. Schrage and Louis W. Miller. The Queue M/G/1 with the Shortest Remaining Processing Time Discipline. *Operation Research*, 14(4):670–684, July–August 1966.

[70] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *Proceedings of the 8th USENIX conference on Networked Systems Design and Implementation (NSDI'11)*, Boston, MA, USA, March 2011.

[71] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation. Technical report, Facebook, Palo Alto, CA, USA, April 2007.

[72] Vijayaraghavan Soundararajan and Jennifer M. Anderson. The Impact of Management Operations on the Virtualized Datacenter. In *Proceedings of the 37th International Symposium on Computer Architecture (ISCA'10)*, Saint-Malo, France, June 2010.

[73] Techcrunch. There Goes The Weekend! Pinterest, Instagram And Netflix Down Due To AWS Outage. http://techcrunch.com/2012/06/30/there-goes-the-weekend-pinterest-instagram-and-netflix-down-due-to-aws-outage/.

[74] Kris Tiri. Side-Channel Attack Pitfalls. In *Proceedings of the 44th annual Design Automation Conference (DAC'07)*, San Diego, CA, USA, June 2007.

[75] Balajee Vamanan, Jahangir Hasan, and T. N. Vijaykumar. Deadline-Aware Datacenter TCP (D^2TCP). In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)*, Helsinki, Finland, August 2012.

[76] Venkatanathan Varadarajan, Thawan Kooburat, Benjamin Farley, Thomas Ristenpart, and Michael M. Swift. Resource-Freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, NC, USA, October 2012.

[77] Vijay Vasudevan, Amar Phanishayee, Hiral Shah, Elie Krevat, David G. Andersen, Gregory R. Ganger, Garth A. Gibson, and Brian Mueller. Safe and Effective Fine-grained TCP Retransmissions for Datacenter Communication. In *Proceedings of the ACM SIGCOMM 2009 conference (SIGCOMM'09)*, Barcelona, Spain, August 2009.

[78] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of the 29th conference on Information communications (INFOCOM'10)*, San Diego, CA, USA, March 2010.

[79] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th International Symposium on Computer Architecture (ISCA'07)*, San Diego, CA, USA, June 2007.

[80] Jon Whiteaker, Fabian Schneider, and Renata Teixeira. Explaining Packet Delays under Virtualization. *SIGCOMM Computer Communication Review*, 41(1):38–44, January 2011.

[81] Wiki. Magnetic Stripe Card. `http://en.wikipedia.org/wiki/Magnetic_stripe_card`.

[82] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2011 conference (SIGCOMM'11)*, Toronto, ON, CA, August 2011.

[83] Timothy Wood, Prashant J. Shenoy, Arun Venkataramani, and Mazin S. Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *Proceedings of the 4th conference on Symposium on Networked Systems Design & Implementation (NSDI'07)*, Cambridge, MA, USA, April 2007.

[84] Jingzheng Wu, Liping Ding, Yuqi Lin, Nasro Min-Allah, and Yongji Wang. XenPump: A New Method to Mitigate Timing Channel in Cloud Computing. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD'12)*, Honolulu, HI, USA, June 2012.

[85] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *Proceedings of the 21st USENIX Conference on Security Symposium (SECURITY'12)*, Bellevue, WA, USA, August 2012.

[86] www.bufferbloat.net. Best Practices for Benchmarking CoDel and FQ CoDel. `http://goo.gl/2RhwY`.

[87] xen.org. Xen 4.2: cpupools. `http://blog.xen.org/index.php/2012/04/23/xen-4-2-cpupools/`.

[88] xen.org. Xen Credit Scheduler. `http://wiki.xen.org/wiki/Credit_Scheduler`.

[89] Sisu Xi, Justin Wilson, Chenyang Lu, and Christopher Gill. RT-Xen: Towards Real-time Hypervisor Scheduling in Xen. In *Proceedings of the 11th International Conference on Embedded Software (EMSOFT'11)*, Taipei, Taiwan, October 2011.

[90] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core. In *Proceedings of the USENIX 2013 Annual Technical Conference (ATC'13)*, San Jose, CA, USA, June 2013.

[91] Cong Xu, Sahan Gamage, Pawan N. Rao, Ardalan Kangarlou, Ramana Kompella, and Dongyan Xu. vSlicer: Latency-Aware Virtual Machine Scheduling via Differentiated-Frequency CPU Slicing. In *Proceedings of the 21st ACM International Symposium on High Performance Distributed Computing (HPDC'12)*, Delft, The Netherlands, June 2012.

[92] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. An Exploration of L2 Cache Covert Channels in Virtualized Environments. In *Proceedings of the 3rd ACM Cloud Computing Security Workshop (CCSW'11)*, Chicago, IL, USA, October 2011.

[93] Yunjing Xu, Michael Bailey, Brian Noble, and Farnam Jahanian. Small is Better: Avoiding Latency Traps in Virtualized Data Centers. In *Proceedings of ACM Symposium on Cloud Computing 2013 (SoCC'13)*, Santa Clara, CA, USA, October 2013.

[94] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, USA, April 2013.

[95] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *Proceedings of the ACM SIGCOMM 2012 conference (SIGCOMM'12)*, Helsinki, Finland, August 2012.

[96] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. HomeAlone: Co-Residency Detection in the Cloud via Side-Channel Analysis. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy (S&P'11)*, Oakland, CA, USA, May 2011.

[97] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS'12)*, Raleigh, NC, USA, October 2012.